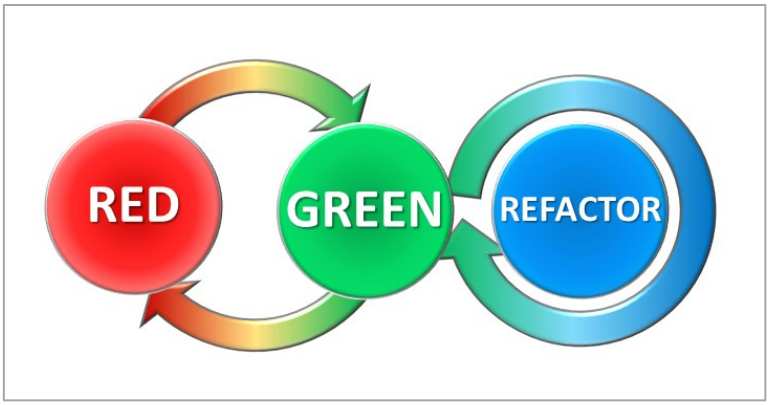
VB2010 Express + NUnit 2.5 で、初めての TDD Step by Step

TDD って、どんなふうに仕事してるのかな?

そんな、あなたに。



http://www.tdd-net.jp/
山本 康彦 (Twitter: @biac)

初版 2011/2/11
(初版) PDF バージョン 2011/2/27

TDD は、テストファーストとリファクタリングだと。 テストファーストは、テストケースを先にコードで表現してから、製品コードを書くのだと。 そんなふうに説明はされるけど、じゃあ**実際にはどうやっているのか?** ごく簡単な Windows 用のプログラムを例題にして、紹介してみます。 なお、ここでは省いていますが、実際にはソースコード管理システム (ソースコード リポジトリ) の扱い方も大切です。

現在の xUnit 系のユニットテスト ツールでは、GUI の自動テストは困難です。 Visual Studio 2010 の上位版では、GUI の自動テストをかなり簡単に作れるようになりましたが、テストファーストでの開発には、まだほど遠いというのが現状です。 したがって、実際の開発では、**GUI 部分は従来からの手法で作成**し、**ロジック部分を TDD で**作っていきます。

このチュートリアルでは、**C++ や Java や C# は知ってるけど、Visual Basic (VB) は初めて**という人でも分かるように、VB の使い方も含めてステップ バイ ステップで解説します。

■ 0. 準備 5
◆ 0-1: VB2010 と NUNIT2.5 5
◆ 0-2: VB の設定について 6
■ 1. 仕様を決める 8
◆ 1-1: 要件 8
◆ 1-2: 画面スケッチ: 8
◆ 1-3: システム分割 9
■ 2. GUI を作る 10
◆ 2-1: ソリューションと製品プロジェクト 10
◆ 2-2: フォーム デザイナー 11
◆ 2-3: イベント ハンドラーを試す 14
§ *TODO* コメントの表示のしかた *16*
■ 3. TDD (テストファースト) 17
◆ 3-1: メソッドの外部設計 17
§ シグネチャー *17*
§ 入出力 *18*
◆ 3-2: テストケースを書く準備 20
§ テスト プロジェクトの追加 *21*
§参照設定 *22*
§テスト クラスの作成 *24*
§*NUnit* の動作確認 *26*
◆ 3-3: 最初のテストケース 29
§ *Assert* から書き始める *29*
§ *result* を得るには*?* *30*
§ 製品クラスを作る *31*
§*InternalsVisibleTo* *33*
§ *SayNext()* の仮実装 *34*
§ ターゲット プラットフォーム *37*
◆ 3-4: TDD 三原則 38
◆ 3-5: ふたつめのテストケース 40
§ レッド (テストケースを書いて失敗させる) *40*

§ グリーン (製品コードを書いて成功させる) 42
◆ 3-6: テストケースのリファクタリング 44
§ テストひとつに *Assert* ひとつの原則 47
◆ 3-7: レッド→グリーンのリズムに乗って 47
§ みっつめのテストケース 47
§ 4 番目のテストケース 49
§ 5 番目のテストケース 50
◆ 3-8: テストファースト終了 52
■ **4. TDD (リファクタリング) 54**
◆ 4-1: 外部設計と内部設計について 54
◆ 4-2: リファクタリングとは 54
◆ 4-3: リファクタリングする候補を挙げる 55
§ 日本語の名前 57
◆ 4-4: リファクタリングを行う 58
§ 変数のリネーム *(Rename)* 58
§ メソッドへ切り出し *(Extract Method)* 59
§ 説明用の変数の導入 *(Introduce Explaining Variable)* 62
§ ふたたび変数のリネーム *(Rename)* 63
§ ふたたび説明用の変数の導入 *(Introduce Explaining Variable)* 63
§ ネストした条件式のガード句による置き換え *(Replace Nested Conditional with Guard Clauses)* 64
§ 一時変数のインライン化 *(Inline Temp)* 65
§ リファクタリング終了 66
§ 付録: *VB* の *Static* キーワード 68
■ **5. システム テスト 69**
◆ 5-1: GUI と結合して、プログラムの完成 69
◆ 5-2: システムテスト 72

■ 0. 準備

◆ 0-1: VB2010 と NUnit2.5

ここでは、Visual Basic 2010 (VB2010) と NUnit 2.x を使います。 Visual Studio 2010 (VS2010) をお持ちでない方は、次の Express Edition (VB2010EE) を利用してください。

- **Visual Basic 2010 Express**
  ダウンロードページ ⇒ **http://www.microsoft.com/japan/msdn/vstudio/express/**
  ※ すべてインストールしても構いませんが、ここで必要なのは Visual Basic 2010 Express だけです。
- **NUnit 2**.5.9 (2011/2/8 現在)
  配布サイト ⇒ **https://launchpad.net/nunitv2** (英語)
  参考記事 ⇒ **NUnit 2.5 の導入 Step by Step** (http://www.tdd-net.jp/nunit-25-step-by-step.html)
  ※ 2011/2 末頃に 2.6.0 がリリースされる予定となっています。 バージョンが上がった場合、ここでの説明と異なる部分が出てくるかもしれません。

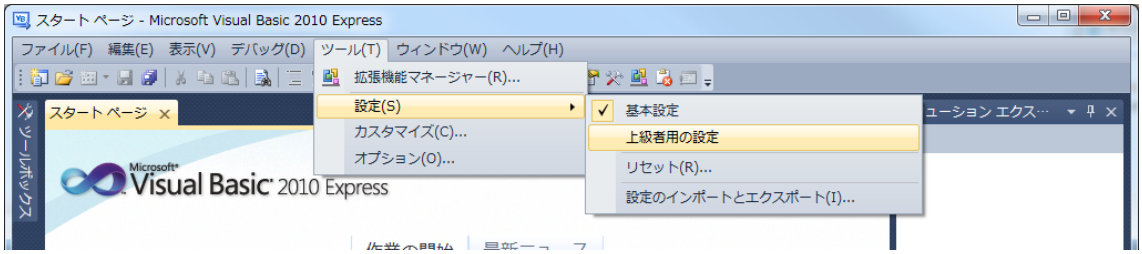
なお、VS2010 の Pro. 版以上には、標準でユニットテスト機能が組み込まれています (MSTest)。 NUnit とテストケースの書き方は異なりますが、考え方は同じです。

◆ 0-2: VB の設定について

VB2010EE をインストールしたら、起動してみましょう。

スタートページ (上部のタブにそう表示されています) の左上にある [新しいプロジェクト...] というリンクをクリックすれば、プログラムを作っていけます。 が、その前に…

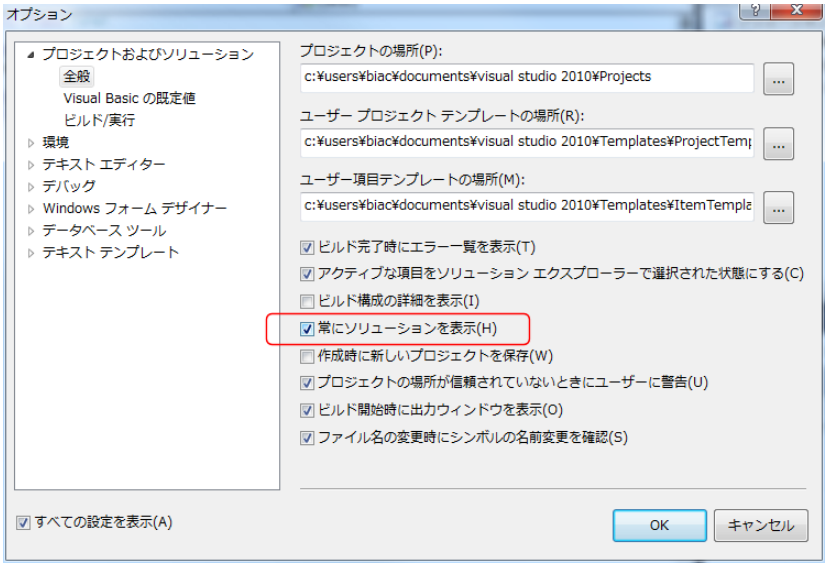
Visual Studio に慣れている方は、Express Edition を使ってみると、その設定の少なさに驚くことでしょう。 必ず次のように設定を切り替えておいてください。



メニューの [ツール] - [設定] - [**上級者用の設定**] を選びます。 すると、オプション設定ダイアログなどに表示される設定項目が増えて、上位版とほぼ同じカスタマイズが出来るようになります。

そうしたら、**次の設定だけは必ず**行っておいてください。

[ツール] - [オプション...] で、オプションダイアログを出します。



ダイアログ左下の 「すべての設定を表示」 にチェックを入れ、左側のツリーで [プロジェクトおよびソリューション] - [全般] を選び、右側の 「常にソリューションを表示」 にチェックを付けます。 これをやっておかないと、複数モジュール (複数プロジェクト) からなるアプリケーションの作成ができません。

■ 1. 仕様を決める

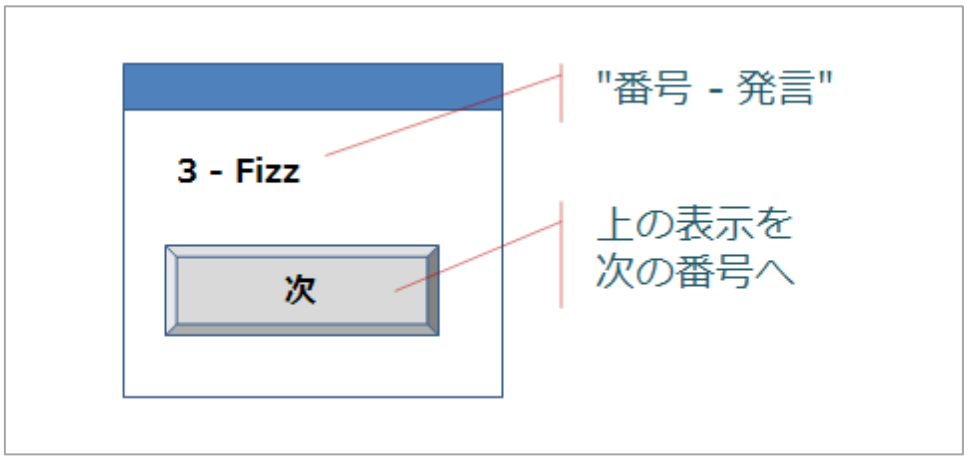
それでは、どんなプログラムを作るか決めましょう。 いわゆる要件定義～概要設計にあたる作業です。 ここでは、テストファーストではお約束となった感のある **"Fizz Buzz"** のプログラムとします。

◆ 1-1: 要件

- 最初に "1 - 1" と表示される。 左は 「番号」 で、右は 「発言」 とする。
- ユーザーが指示するごとに、Fizz Buzz のルールに従って次々に表示が変わる。
- Fizz Buzz のルールは次の通り。
  1. 最初は番号=1、発言="1"
  2. 次々と番号を 1 ずつ増やしていく。 ただし発言は、
     - 2a. **3 で割り切れる場合は "Fizz"**
     - 2b. **5 で割り切れる場合は "Buzz"**
     - 2c. **3 でも 5 でも割り切れる場合は "Fizz Buzz"**
     - 2d. **それら以外は数字**
- 番号の上限は Int32 の最大値まで。

◆ 1-2: 画面スケッチ:

図のような Windows フォームとします。



※ 省略していますが、もちろん右上に [X] (閉じる) ボタンがあります。

◆ 1-3: システム分割

簡単なプログラムなので、製品コードで 1 モジュール (exe ひとつ)、ユニットテストで 1 モジュール (dll ひとつ) とします (合わせて 2 モジュール = 2 プロジェクト)。 また、ふたつのプロジェクトをバラバラに開発するよりも、Visual Studio の 「ソリューション」にプロジェクトを入れて管理した方が便利なので、そのようにします。

ソリューション "FizzBuzzByTDD"
+ FizzBuzz.exe：Windows フォームのプロジェクト "FizzBuzz" (製品コード)
+ FizzBuzzTest.dll：クラスライブラリーのプロジェクト "FizzBuzzTest" (テストコード)

なお、ちょっとした規模 (画面が数枚以上) のアプリケーションでも、このプロジェクト分割と、プロジェクト間の依存関係については、事前にしっかり検討しておきましょう。

製品コード FizzBuzz.exe の中も、簡単なプログラムですから、UI とロジックの 2 つに分けるだけでおそらく足りるでしょう。

FizzBuzz.exe
+ Form1.vb (UI のクラス)
+ FizzBuzzer.vb (ロジックのクラス)

どのようなクラスが必要になるかは、事前にはなかなか分からないものです。 ここは大雑把に検討しておき、実装を進める中でプロジェクトの中にフォルダーを作って分かりやすく配置しなおしたりしていきます (このチュートリアルでは行いません)。 ひとつのフォルダー内に 10 個もソースファイルが入っていたら、フォルダーに分けることを考えましょう。 ひとつのプロジェクトがあまりにも肥大化したときには、プロジェクト (モジュール) を分割するという大手術をすることもあります。

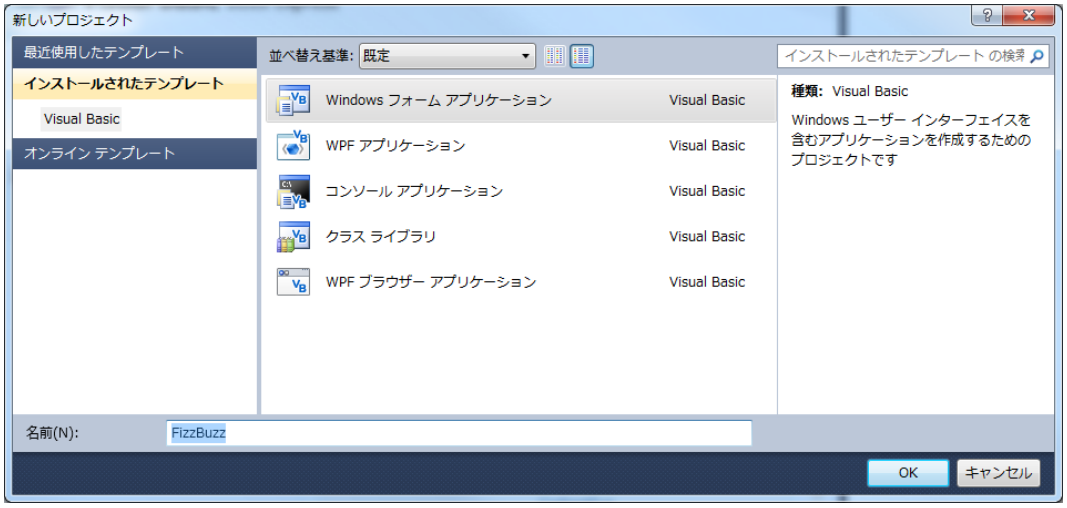
■ 2. GUI を作る

◆ 2-1: ソリューションと製品プロジェクト

それではプログラムを作り始めましょう。

※ このあたりは分かってるよという方は、前述のプロジェクト構成と画面を作り、ボタンのクリック イベント ハンドラーも作成してしまって、「3. TDD (テストファースト)」 へ進んでください。

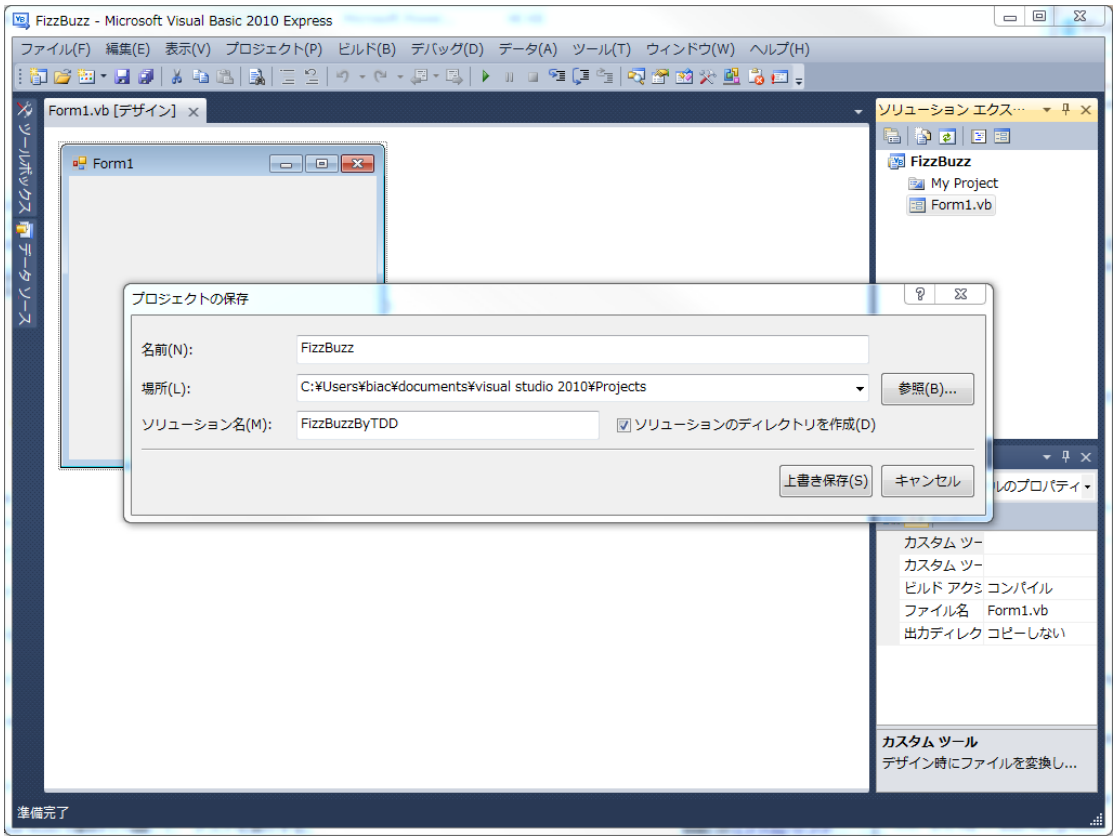
VB2010 を起動し、スタートページの [**新しいプロジェクト...**] リンクをクリックするか、メニューの [ファイル] - [新しいプロジェクト...] を選択します。 すると、「新しいプロジェクト」 ダイアログボックスが出てきます。



画像にあるように 「Windows フォーム アプリケーション」 (Windows フォームを GUI に使う exe ファイルを作成するためのプロジェクト) を選択し、下の 「名前」 の欄にはプロジェクト名 "FizzBuzz" を入力して [OK] ボタンをクリックします。

しばらくして VB2010 のプロジェクト作成処理が終わると、フォーム デザイナー画面が開きます。 ただし VB2010 では、この状態でのソースコードは一時的な場所に置かれています。 ちょっと試したいときはそれでもよいのですが、これからしばらくお付き合いするコードですから、分かる場所に保存しなおしておきます。

新しくプロジェクトを作ったら、メニュー [ファイル] - [すべてを保存] を実行します。すると 「**プロジェクトの保存**」 ダイアログが出てきます。



最初の 「名前」 欄はプロジェクト名で、ここで変更することもできます。 「場所」 は、ソリューションを置くフォルダを指定します。 「ソリューション名」 には、ソリューションに付ける名前 (ここでは "FizzBuzzByTDD") を指定します。 その右の [ソリューションのディレクトリを作成] にチェックを付けると、「場所」 で指定したフォルダの中に「ソリューション名」 のフォルダが作られて、そこにソリューションが配置されます。はじめての場所に保存するのに [上書き保存] というのは不思議ですが、その [上書き保存] ボタンをクリックすると指定した場所にソリューソンとプロジェクトが保存されます。

保存出来たらそのフォルダを Windows Explorer で見て、どのようなフォルダ構成とソース ファイルが作成されたか確認してみてください。 また、VB2010 のタイトルバーの左端の表示が、プロジェクト名 (FizzBuzz) から、ソリューション名 (FizzBuzzByTDD) に変わっています。

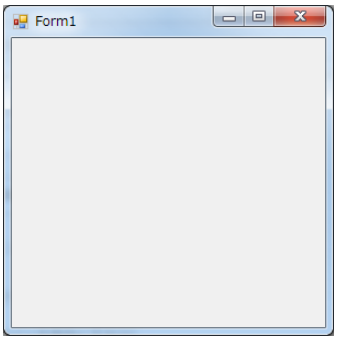
◆ 2-2: フォーム デザイナー

Windows フォーム プロジェクトを作成すると、フォーム デザイナー画面が自動的に開きます (タブに "Form1.vb [デザイン]" と表示されている)。 開いていないときは、右側

の 「ソリューション エクスプローラー」 で Form1.vb をダブルクリック (あるいは、右クリックで [デザイナーの表示]) してください。 上の画像のように、"Form1" というタイトルが付いた、枠だけのウィンドウが表示されます。

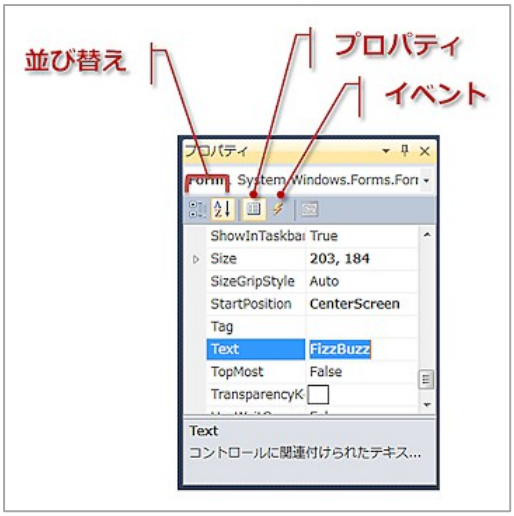
このようにして Visual Studio に何かを自動生成させたときは、とりあえずそのままビルド・実行して、どんなものを生成してくれたのか確認してみましょう。 メニューの [デバッグ] - [デバッグ開始] (あるいは F5) で、ビルドされ、コンパイル エラーが無ければ**デバッグ実行**されます。

デバッグ実行すると、このように中身が何もないウィンドウが表示されます。 それでも上部のタイトルバーの部分の機能は備わっていて、左端のコントロールボックス、右側に並んだ最小化ボタン・最大化ボタン・閉じるボタンは動作します。



閉じるボタンでデバッグ実行を終了させたら、前に 「1-2: 画面スケッチ」 で書いたように、画面のデザインを作っていきましょう。

フォーム デザイナーの画面で主に使うのは、普段は左端に格納されている 「**ツールボックス**」 と右側の 「**プロパティ**」 というペイン (画面領域を区切った一部) です。



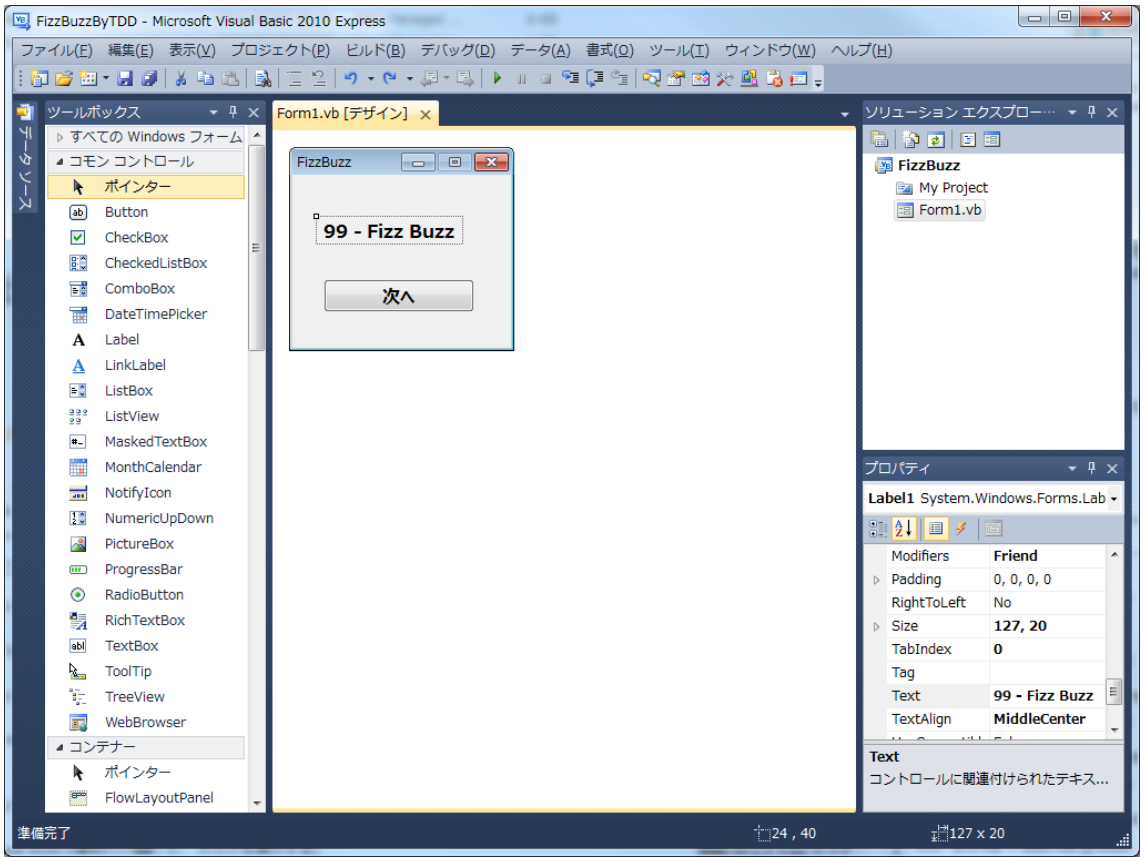
まずはフォームのタイトルを変えてみましょう。 メインのペインで、フォームのタイトルバー ("Form1" と表示されている部分) をクリックして、フォームが編集対象になった状態にします。 すると、プロパティ ペイン (左図) には Form1 のプロパティが表示されますので、その一覧の中から Text プロパティを探して "Form1" を "FizzBuzz" に変更します。

続けて (フォームが編集対象になった状態のまま)、ツールボックスの Label と Button をそれぞれダブルクリックして、コントロール (部品) をフォームに貼り付けます。

※ 貼り付け方は、他に 2 通りあります。 ツールボックスからドラッグ&ドロップする方法と、ツールボックスで選択したらマウスボタンを一旦離してからフォーム上でマウスをドラッグする方法です。

フォームに貼り付けた Label と Button を適当な位置にマウスで移動させ、プロパティを設定していきます。

- Button の Name プロパティは、**ButtonText** にする。
- Label のテキストプロパティに "99 - Fizz Buzz"、Button の Text プロパティを "次へ" とすると、次のようなフォームになります。 (画像では、その他のプロパティも設定してありますが、この後の作業に違いはありません。)



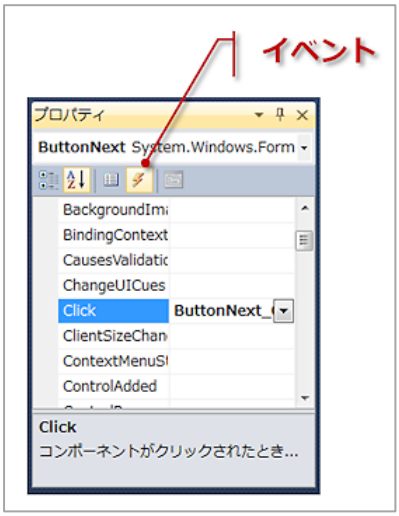
一通り配置とプロパティ設定が終わったら、またデバッグ実行してみて UI のデザインを確認します。

※ ちなみに上の画像は、「0-2: VB の設定について」 で記述した 「常にソリューションを表示」 の設定を行っていない状態です。 「ソリューション名の指定」 だの、「ソリューション エクスプローラー」だのと言ってきましたが、VB2010EE のデフォルト状態では、ご覧のようにソリューションエクスプローラーにソリューション (ここでは "FizzBuzzByTDD") は表示されず、プロジェクト (ここでは "FizzBuzz") だけが表示されます。

※ なお、このようにコントロールを適当に貼り付けただけの画面では、「製品」 になりません。 実行して、フォームの拡大縮小してみてください。 また、Windows の DPI 設定を変えたらどうなるでしょう? そのあたりは本稿の趣旨から外れすぎますので割愛しますが、実際の開発では必要なことです。

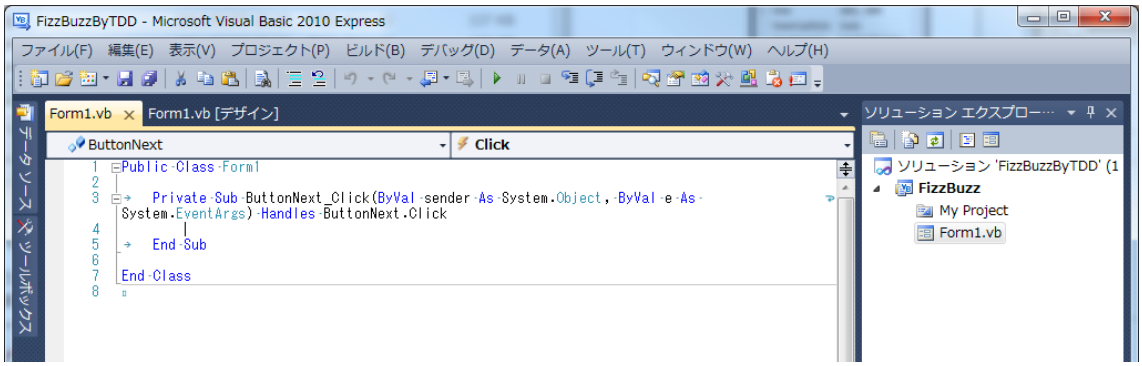
◆ 2-3: イベント ハンドラーを試す

コントロールを配置しただけでは、ボタンをクリックしても何も起きません。 クリックされた時に呼び出されるメソッド (イベント ハンドラー) を設定し、仮の実装をしてみます。



ボタンがクリックされた時のイベント ハンドラーを生成するには、デザイナー画面でそのボタンをダブルクリックします。 または、デザイナー画面でそのボタンを選択状態にしておいて、プロパティ ペインでイベントを指定して、Click の右側の空欄 (左図ではハンドラーを生成した後なので "ButtonNext_Cli…" となっている部分) をダブルクリックします。

すると、ボタンがクリックされた時に呼び出されるイベント ハンドラーが生成され、次の画像のようにコード エディタが自動的に開かれて、生成されたイベント ハンドラーにカーソルが置かれます。



この生成されたイベント ハンドラー ButtonNext_Click() メソッドの中身は空っぽですから、デバッグ実行してボタンをクリックしてみても、なにも起きません。 このメソッドには、何をやってもらえばいいでしょう? Fizz Buzz の次の 「番号」 に該当する 「発言」 を表示してもらえばよいのでしたね。 次の 「発言」 を決定する処理はこれから TDD で作るとして、表示するところだけ仮に実装しておきます。 そうですね、ラベルに "TEST" と表示することにしましょうか。 するとイベント ハンドラーの仮実装は次のようになります。

【イベント ハンドラーの仮実装】

```
Private Sub ButtonNext_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles ButtonNext.Click
    'TODO: 後でロジックの呼び出しに置き換える
    Me.Label1.Text = "TEST"
End Sub
```

行のアポストロフィ 「'」 以降はコメントです。

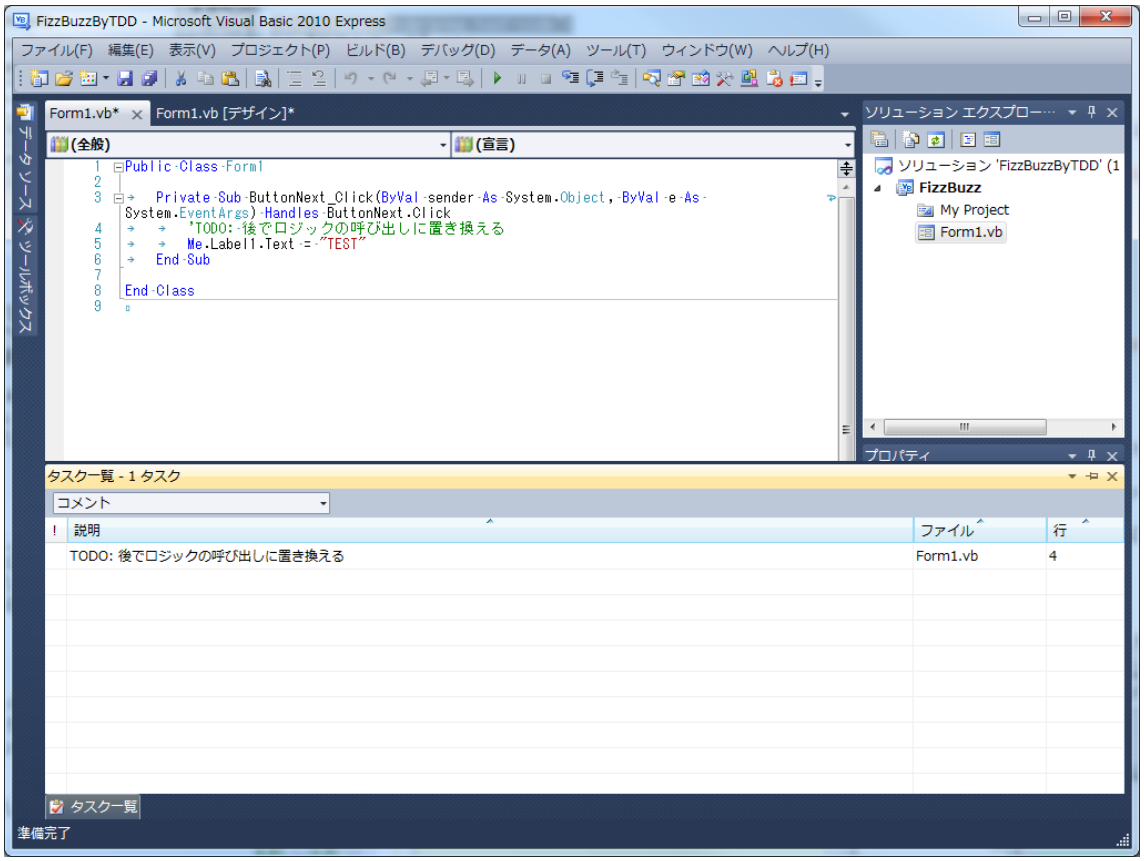
Me キーワードは、C++ や Java の this です。

デザイナー画面で Label1 の Text プロパティに "99 - Fizz Buzz" と設定しましたが、こんどは "TEST" と設定することをコードでやっています。

なお、コメントの最初に 「TODO:」 と書いてあるのは、俗に 「TODO コメント」 と呼び、次のようにして後から簡単に一覧できます。

§ TODO コメントの表示のしかた

タスク一覧ペインを見ます。 表示されていない場合は、メニューの [表示] - [その他のウィンドウ] - [タスク一覧] で出てきます。 次に、タスク一覧ペインの左上にあるドロップダウンボックスで [コメント] を選択すると、TODO コメントの一覧に切り替わります。



それではデバッグ実行して、ボタンをクリックしてみてください。 "99 - Fizz Buzz" の表示が "TEST" に切り替わりましたね。

☆ この時点でのソースコード一式 ⇒ 「FizzBuzzByTDD01.zip」 (12,261 バイト) をダウンロード http://www.tdd-net.jp/files/FizzBuzzByTDD01.zip (VB2010EE 用、VWD2010 や VS2010 上位版でも可)

■ 3. TDD (テストファースト)

それでは、『Fizz Buzz の次の 「番号」 に該当する 「発言」』 を求めるロジックを、テストファーストで作っていきましょう。

◆ 3-1: メソッドの外部設計

といっても、はやる気持ちを抑えて、まずは外部設計からです。

慣れてくれば脳内だけで OK ですし、そうなってくると、最初に大雑把に考えておいて、あとはテストケースを書きながら考えていけるようになります。 ここは初めてのチュートリアルですから、先にきっちり外部設計をやっておきましょう。

ロジックと言いましたが、それはイベント ハンドラーから呼び出されるメソッドです。つまり、メソッドの外部設計ということですが、それはどういうことでしょう? メソッドを外から見たときに、何が決まっていればよいでしょう?

メソッドの外部設計は、基本的には次のふたつの事を決めます。

- シグネチャー: メソッド名、引数、返値、アクセス レベル
- 入出力: 可能な引数 (入力) のパターンに対し、どんな返値 (出力) が返ってくるか?

※ 入力には、引数の他に、返値に影響を与えるクラスのメンバー変数や外部データ (設定ファイルやデータベース、あるいは通信によって得られるデータなど) も含みます。 今回の Fizz Buzz には、そのような外部データはありません。 また出力には、データベースの変更や通信への出力なども含みますが、これも今回の Fizz Buzz にはありません。

※ なお、入出力がそのメソッドの属するクラスの状態に依存する場合は、先にクラスの状態遷移を決める必要がある場合もあります。

§ シグネチャー

一番大切なのは、**メソッドの名前**です。 UI から FizzBuzzer クラスに対して、「次に表示すべきことを教えてくれ!」 と依頼するためのメソッドですから、**SayNext()** と名付けましょうか。

引数ですが、これはありません。

返値は、メソッドを呼び出すごとに変わっていきます。 最初は "1 - 1"、次に呼び出すと "2 - 2"、その次は "3 - Fizz" と。 そうそう、返値をそのまま UI のラベルの文字列にセットできると楽なので、返値の型は **string** でしょうね。

アクセス レベルは、同じモジュール (プロジェクト) 内の Form1 から見えれば良いです。 これは VB では **Friend** と表記します。

※ ユニットテストの都合でアクセス レベルを変えることは、よくあります。 ここでは Friend スコープで大丈夫です。

以上をまとめると、シグネチャーは次のようになりました。 なお、VB では、返値がある場合は Function (関数) であると宣言し、無い場合は void ではなく Sub (サブルーチン) であると宣言し分ける必要があります。 また、返値の型は、メソッド名の前ではなく、後ろに As キーワードを付けて記述します。

```
    Friend Function SayNext() As String
```

また、このメソッドを呼び出す UI のイベント ハンドラーは、次のような形になるはずです。

【イベント ハンドラーの実装 (予定)】

```
Private Sub ButtonNext_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles ButtonNext.Click
    Me.Label1.Text = hoge.SayNext()
End Sub
```

§ 入出力

SayNext() メソッドは引数を取りません。 しかし出力は、呼び出すたびに変わります。それはもちろん、FizzBuzzer クラスのメンバー変数か何かに 「番号」 を保持しているから出来ることですが、クラスの内部構造 (内部設計) を先に固定してしまいたくはありません (実際には外部設計段階でそうせざるをえない場面も、往々にしてありますけれど)。

ここでは、何回目の呼び出しであるかということと結びつけて、出力を定義しましょう。この定義ならば、SayNext() メソッドをテストすることができます (プライベートなメンバー変数を入力としてしまうと、簡単にはテストコードが書けない)。

出力を定義するにあたって、定性的な定義は日本語で要件定義に書きました。 それを厳密な数式で表現できるならばよいのですが、そうでないとき (あるいは、困難な時など) は同値分割を行って代表ケースを洗い出しておきます。

※ Fizz Buzz は数式表現が可能ですが、ここでは同値分割を行っておきます。

※ テスト的な観点から境界値をたくさん載せていますが、じつは TDD 的には不要です。 後ほど説明する TDD 三原則をご覧ください。

呼び出し回	同値クラス	返値
1 回目	(d) 3 の倍数でない 5 の倍数でない	"1 - 1"
2 回目	(d) 3 の倍数でない 5 の倍数でない	"2 - 2"
3 回目	(a) 3 の倍数 5 の倍数でない	"3 - Fizz"
4 回目	(d) 3 の倍数でない 5 の倍数でない	"4 - 4"
5 回目	(b) 3 の倍数でない 5 の倍数	"5 - Buzz"
6 回目	(a) 3 の倍数 5 の倍数でない	"6 - Fizz"
7 回目	(d) 3 の倍数でない 5 の倍数でない	"7 - 7"
14 回目	(d)	"14 - 14"

	3 の倍数でない 5 の倍数でない	
15 回目	(c) 3 の倍数 5 の倍数	"15 - Fizz Buzz"
16 回目	(d) 3 の倍数でない 5 の倍数でない	"16 - 16"
Integer.MaxValue 回目	(d) 3 の倍数でない 5 の倍数でない	"2147483647 - 2147483647"
(Integer.MaxValue + 1) 回目	(?)	(?)

…はて?

要件の最後 「番号の上限は Int32 の最大値まで」 に従って Integer.MaxValue 回目のケースを書き、さらに (Integer.MaxValue + 1) 回目のときを書こうとしましたが、そのときの出力はどうしましょう? Integer.MaxValue 回のときに、もう一回ボタンをクリックしようとしたらどうなるか、要件の検討の時に考えるのを忘れていたわけです。

このようなとき実際の開発では、要件検討に差し戻しです。 顧客と合意を取った出力を定義しないといけません。 ここはチュートリアルですから、即決できます。 例外が出てプログラムは落ちちゃうことにしましょう。

(Integer.MaxValue + 1) 回目	(e)	※ 例外発生。プログラム停止

◆ 3-2: テストケースを書く準備

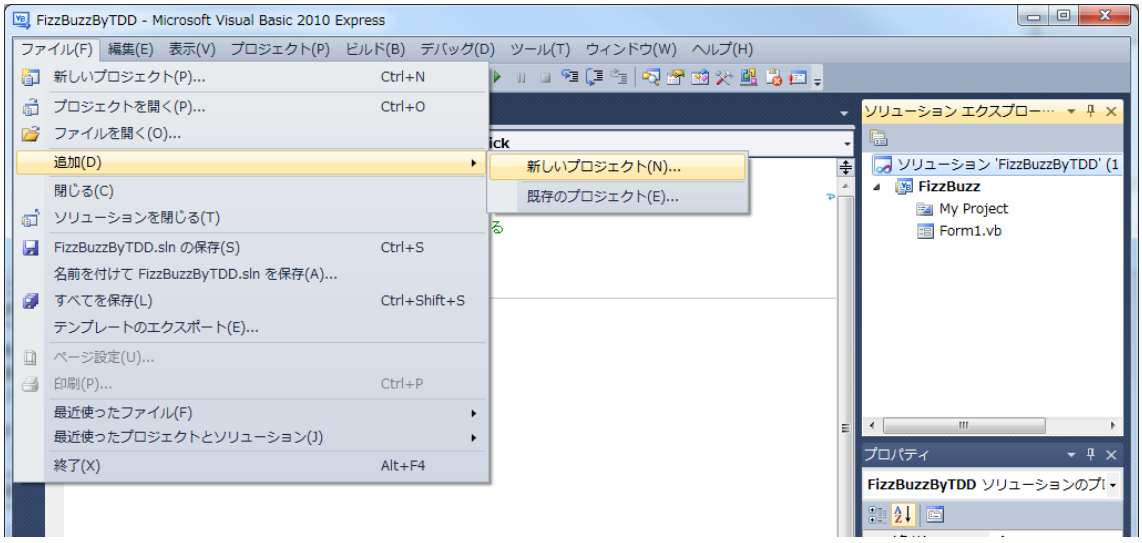
お待たせしました、ようやくテストファーストを始めます …と言いたいところですが、その前にしなければならない準備作業がけっこうあります。

§ テスト プロジェクトの追加

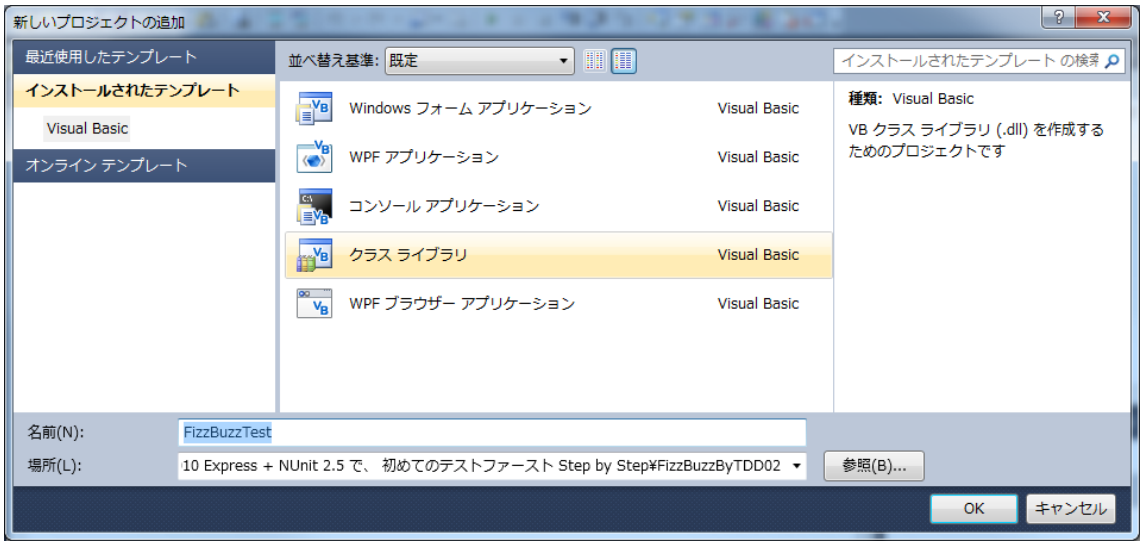
「1-3: システム分割」 で書いたように、今回のテスト コードは製品コードとは別のプロジェクトにします。 .NET Framework を使った開発では、このように製品とテストを分けることが多いようです。

※ 同一プロジェクトにテスト コードを入れると、製品出荷時にテストを取り除くのが面倒になる。

既存のソリューションに新しいプロジェクトを追加するには、VB2010 のメニュー [ファイル] - [追加] - [**新しいプロジェクト...**] を使います (または、ソリューション エクスプローラーでソリューションを右クリックして、[追加] - [新しいプロジェクト...])。



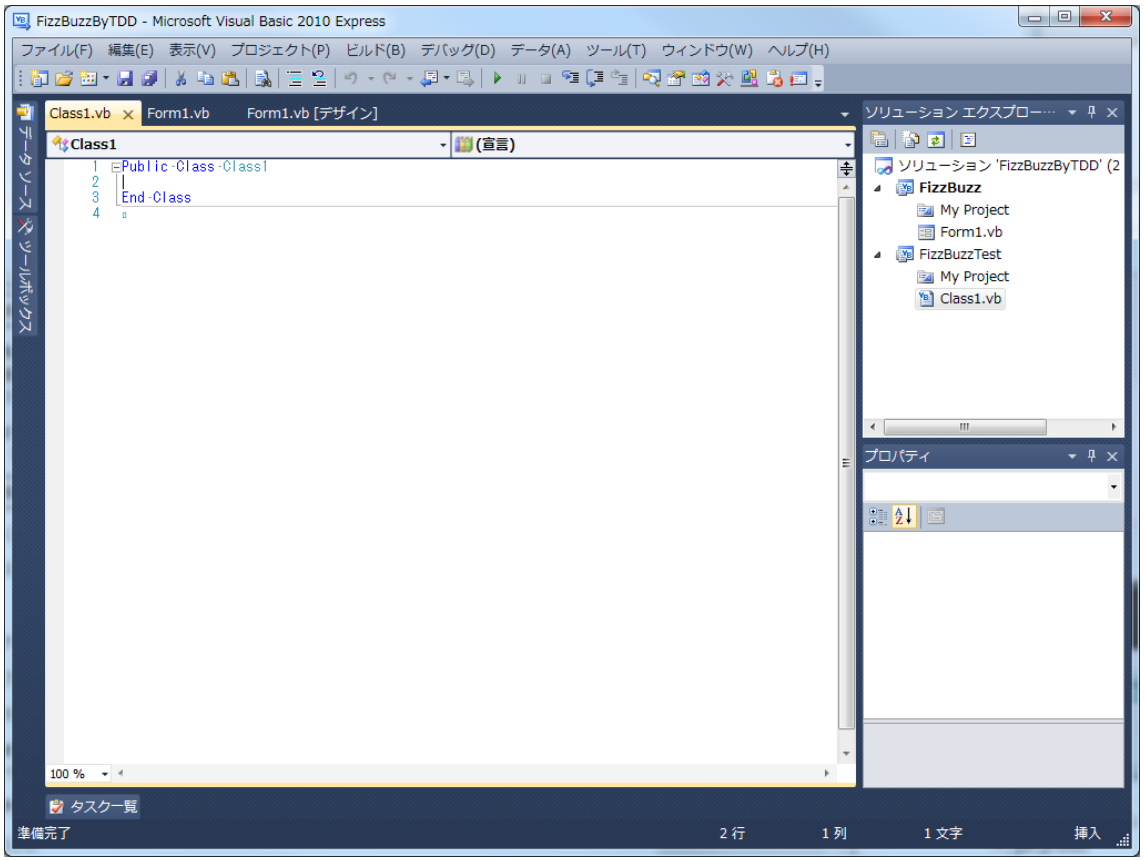
出てきた 「新しいプロジェクトの追加」 ダイアログで、「**クラス ライブラリ**」 を選びます。 テスト コードは、NUnit のランナー プログラムから呼び出して実行するために、dll として作るわけです。



ダイアログの 「名前」 欄には、新しく作るプロジェクトの名前を入力します。 ここでは "FizzBuzzTest" とします。 製品コードのプロジェクト名の後ろに、"Test" や "Tests" などと名付けるのが一般的です。

プロジェクトの名前と同じ名称のフォルダーが作成され、その中にソースコードが置かれます。 フォルダーが作成されるのは、ダイアログの 「場所」 テキストボックスに入力したフォルダーの中です。

[OK] ボタンをクリックすると、"FizzBuzzTest" プロジェクトと Class1.vb という仮のソース ファイルが生成され、画面は次のようになります。



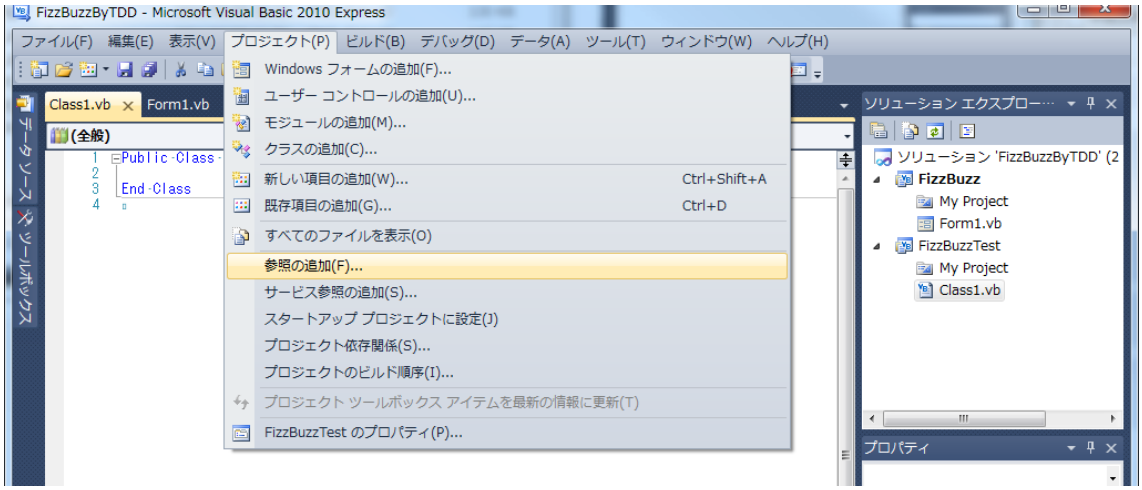
§ 参照設定

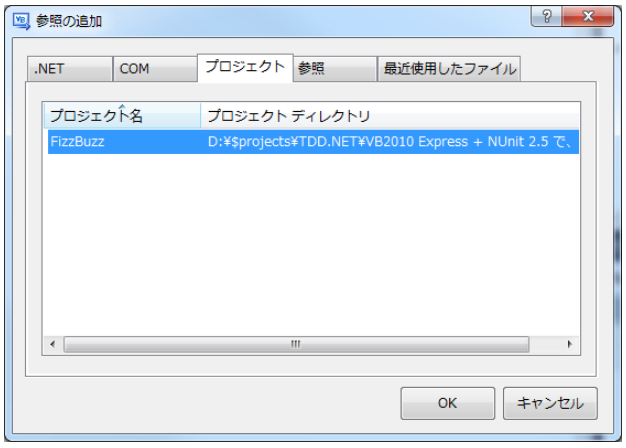
新しく出来たプロジェクトに、参照設定をします。 参照設定には、次のような意味があります。

- コンパイル時に、参照先のモジュールを 「リンク」 する。 (実際には、リンカープログラムでひとつにまとめるわけではないです)

- コーディング時に、Visual Studio が参照先も見てくれる。 (自動補完などの対象になる。 逆に、参照設定が足りないとエラー報告が出る。)

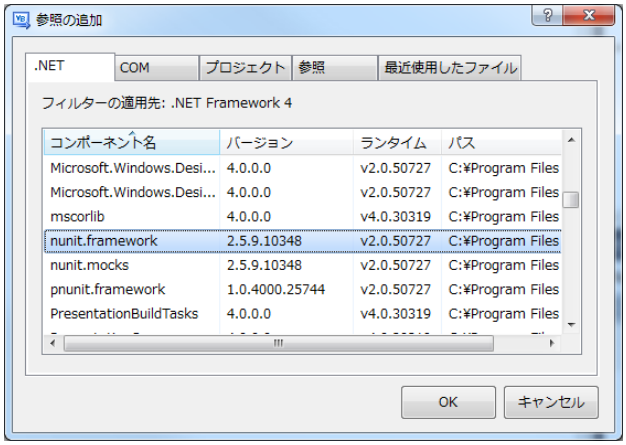
プロジェクトに参照設定を追加するには、ソリューション エクスプローラーで参照する側のプロジェクト (FizzBuzzTest) か、あるいはその中のソース ファイルを選択しておいて、VB2010 のメニュー [プロジェクト] - [**参照の追加...**] を使います (または、ソリューション エクスプローラーで参照する側のプロジェクトを右クリックして、[参照の追加...])。





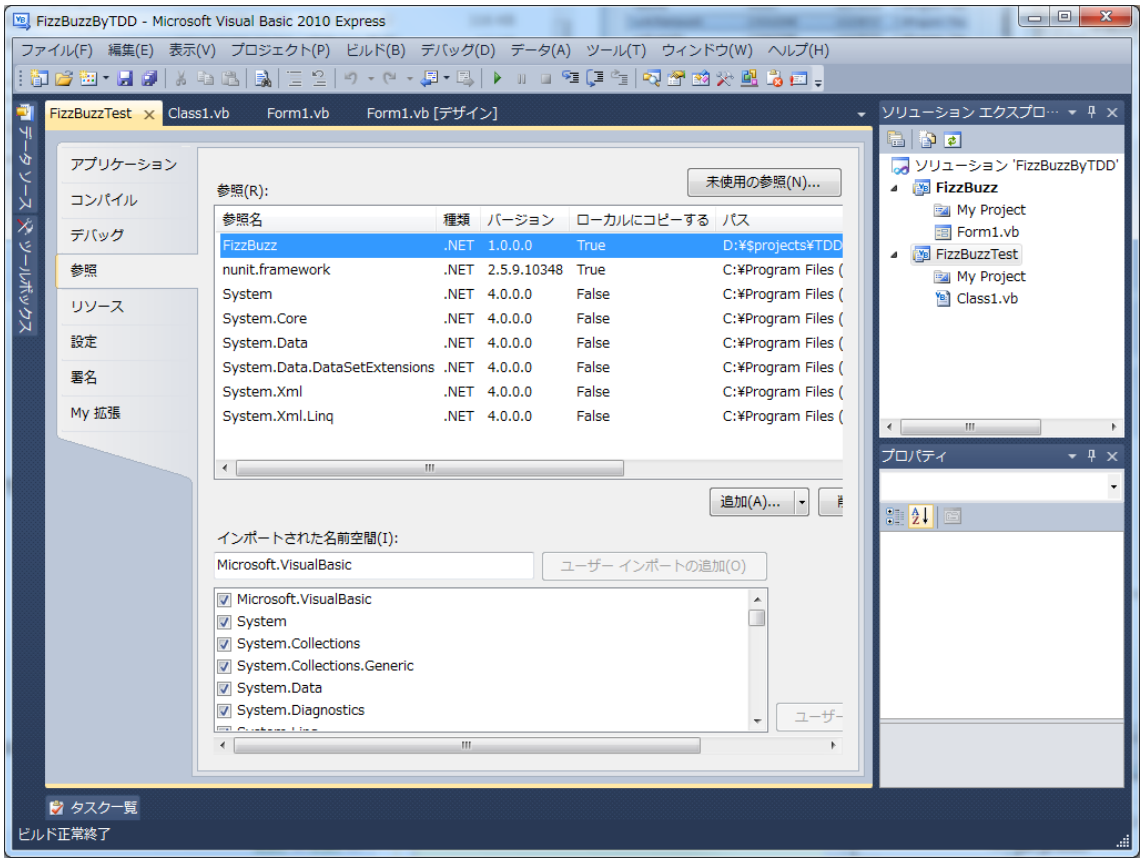
ここでは、2 つの参照設定を追加する必要があります。 まずひとつめは、出てきた [参照の追加] ダイアログで [**プロジェクト**] タブを選んで、製品コードである [**FizzBuzz**] プロジェクト。これを選んだら [OK] ボタンで一旦元の画面に戻ります。

もうひとつは、再び [参照の追加] ダイアログを開いて、[**.NET**] タブを選び、[**nunit.framework**]。



※ 複数バージョンの NUnit がインストールしてある場合は、バージョンとランタイムに気を付けてください。

なお、参照設定の状態は、そのプロジェクトのプロパティ画面の [参照] タブで確認できます。 (ここから追加することも可能)



§ テスト クラスの作成

次に、テスト ケースを書いていくクラスを準備します。 テスト対象のクラスは、「1-3: システム分割」 に書いたように "FizzBuzzer" ですから、テストする側のクラスは

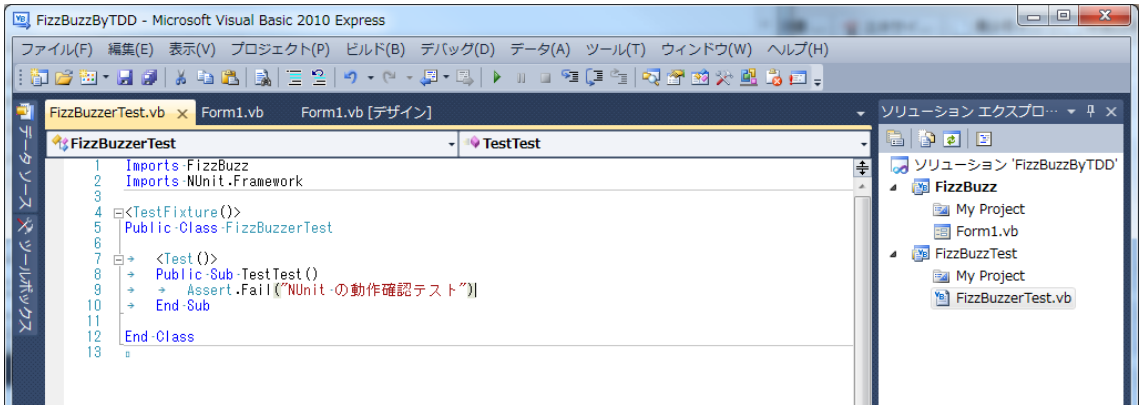
"**FizzBuzzerTest**" としましょう。 自動生成された Class1.vb の名前を変えて使うことにします。

ソリューション エクスプローラーで "Class1.vb" を右クリックし [名前の変更] を選ぶ (あるいは、"Class1.vb" をゆっくり 2 回クリックする) と、ファイル名を編集できるので、"FizzBuzzerTest.vb" に変えます。

※ VB2010 では F2 キーは別の機能に割り当てられています。 なんてこった!

ファイル名を変えると、(デフォルトでは) 自動的にそこに含まれているクラス名も変更されます。 FizzBuzzerTest.vb のコードを見て、クラス名が FizzBuzzerTest に変わっていることを確認しておいてください。 また、このあたりで一旦ビルドして、コンパイルエラーが出ないことを確認しておきます。

それでは、FizzBuzzerTest.vb を開いて、次のようにコードを入力します。



【テストクラス】

```
Imports FizzBuzz
Imports NUnit.Framework


<TestFixture()>
Public Class FizzBuzzerTest


    <Test()>
    Public Sub TestTest()
        Assert.Fail("NUnit の動作確認テスト")
    End Sub


End Class
```

先頭の 2 行は、名前空間のインポート指定です。
クラスやメソッド宣言の前に付けた <TestFixture()> などは属性の宣言です。 C# なら角カッコを使うところ。 NUnit は、TestFixture 属性を見て、そのクラスにテスト ケースが入っていることを、また Test 属性を見て、そのメソッドがテスト ケースであることを、認識します。

Assert クラスは NUnit のものです。 テストケースを書くには、Assert クラスが必須です。 その Fail() メソッドは、それが実行されるとテストは失敗したことになるというもの。 したがって、NUnit からこの TestTest() というメソッドを実行すると、テストは失敗するはずです。

ビルドしてエラーが無いようでしたら、次へ進みましょう。

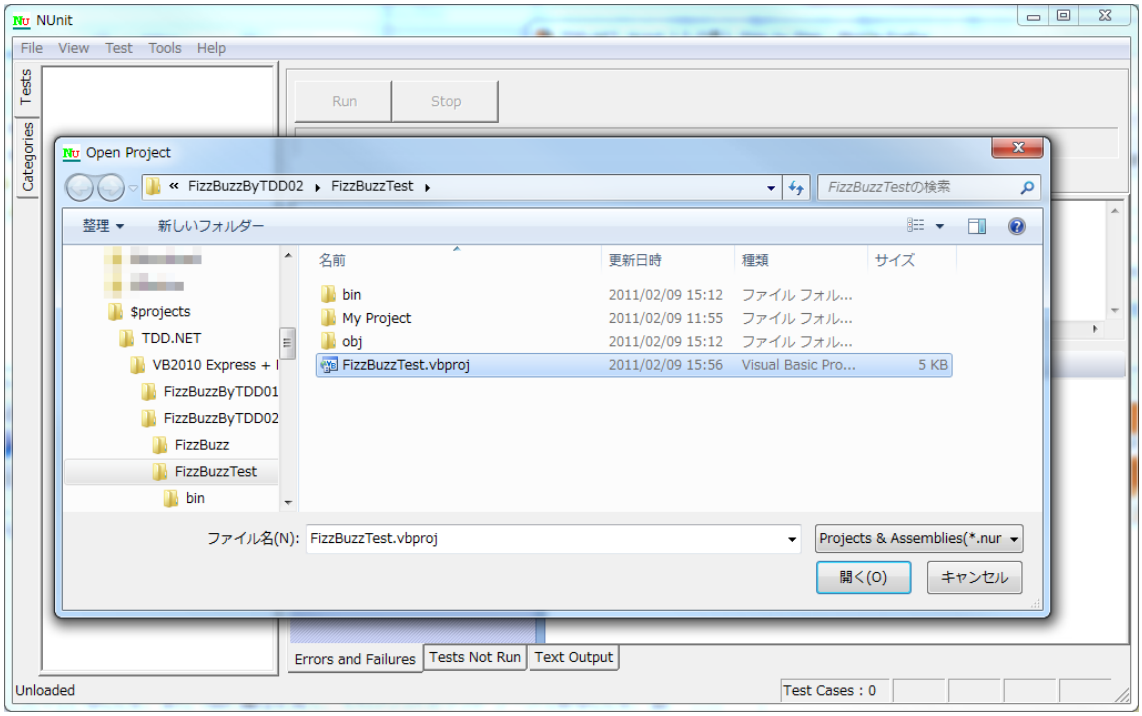
§ NUnit の動作確認

ようやく NUnit の出番です。

NUnit のテスト ランナーには、コンソール版と GUI 版がありますが、GUI の方を使いましょう。

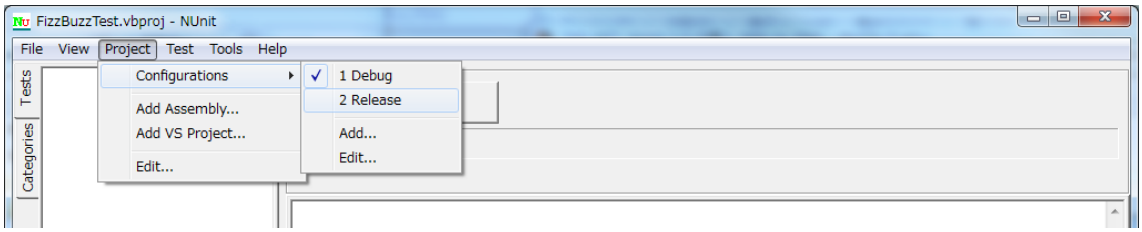
※ オプション [Enable Visual Studio Support] を ON にしておいてください。 詳しくは、「**NUnit 2.5 の導入 Step by Step**」(http://www.tdd-net.jp/nunit-25-step-by-step.html) を参照。

NUnit を起動したら、メニュー [File] - [**Open Project...**] を選びます。 「Open Project」 ダイアログが出てくるので、FizzBuzzTest のプロジェクトファイル "**FizzBuzzTest.vbproj**" を指定して開きます。

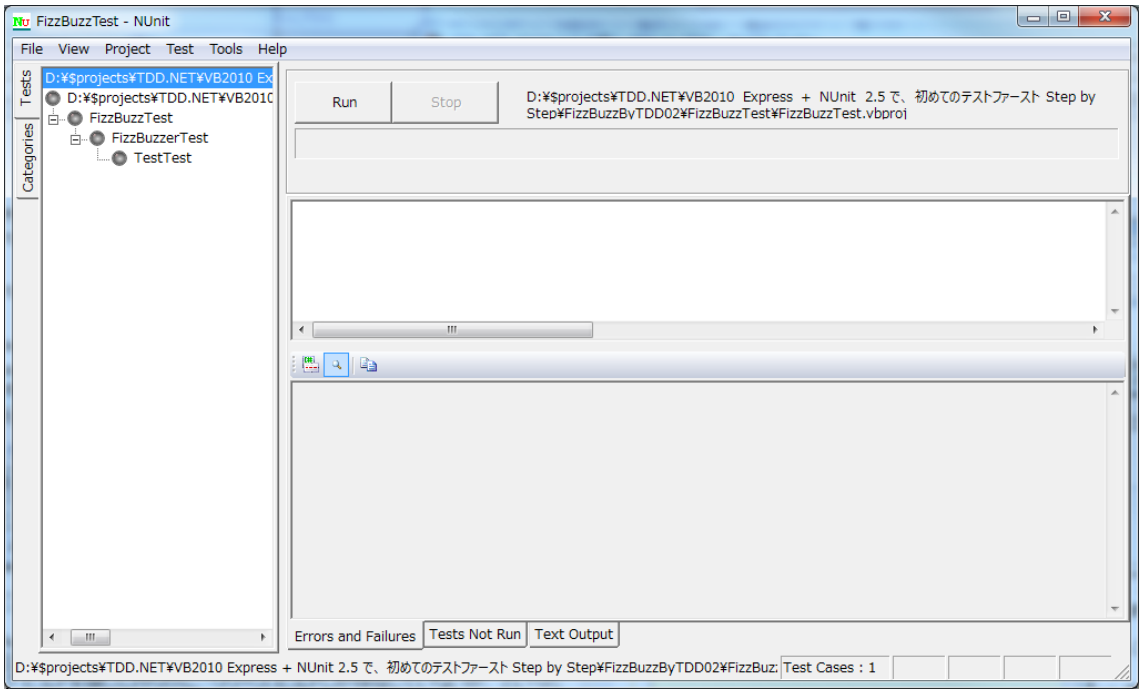


このとき、"Assembly No Loaded" 「…FizzBuzzTest.dll' が見つかりません…」 というエラーになることがあります。 これは、まだビルド出来ていないか、あるいは、NUnit はデフォルトでデバッグ ビルドを探しに行くのに、リリース ビルドしか存在していないか、といったことが原因です。

VB2010EE のデフォルトは**リリース ビルド**です。 この場合、NUnit の設定を変えて、リリース ビルドの方を読み込み直させます。 エラー ダイアログを閉じたら、メニュー [Project] - [Configrations] - [Release] を選びます。



これで NUnit に FizzBuzzTest が読み込まれます。

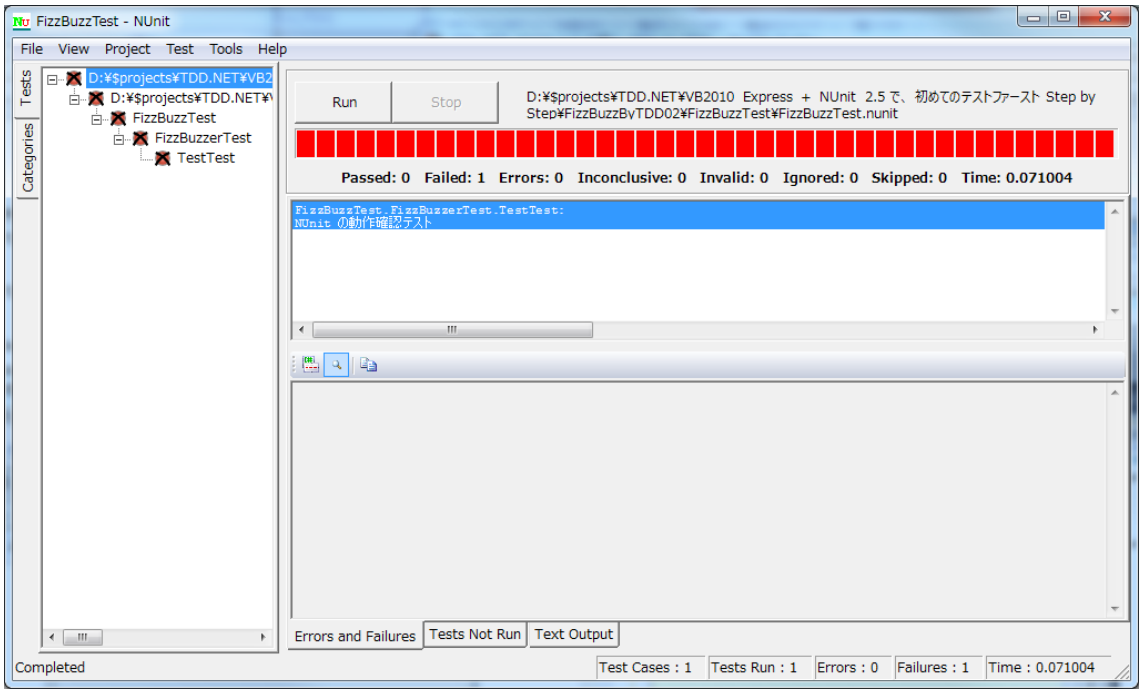


忘れないうちに、NUnit の構成ファイルとして保存しておきましょう。 メニュー [File] - [Save as...] でファイル名を付けて保存します。 デフォルトでは "FizzBuzzTest.nunit" になります。 次からはこの "FizzBuzzTest.nunit" を開けば、FizzBuzzTest を読み込んだ状態で NUnit が起動します。

さて、NUnit はちゃんと動くでしょうか?

FizzBuzzTest の中の TestTest を実行すると、その中身は Assert.Fail() でしたから、失敗するはずです。 Assert.Fail() の引数に指定した "NUnit の動作確認テスト" というメッセージも表示されるはずです。

[**Run**] ボタンをクリックしてテストを実行してみましょう。



予想通りに失敗しました。 ちゃんと NUnit は動作しています。

◆ 3-3: 最初のテストケース

ようやくテストケースを書き始められます。 (ほんとうに 「ようやく」 です。 Fizz Buzz のような単純なサンプルだと、実際に準備にかかる時間のほうが長いくらいです。)

その前に。 準備のために使っていた頭を、テストケースを考える頭に切り替えるために、「3-1: メソッドの外部設計」 のところをざっと眺めて思い出しておいてください。

では、最初のテストケースはどれをやりましょう?

簡単なケースから選びましょう。 1 回目は "1 - 1" が返るというケースです。

§ Assert から書き始める

NUnit の動作確認のために書いた TestTest() メソッドは消してしまって、新しくテスト メソッドを書き始めます。 なにが断言 (assert) できれば、最初のテストは OK だと言えるでしょう? 返ってきた結果が "1 - 1" と等しければ OK ですね。

【最初のテストケース － 記述中(実行不可)】

```
<Test()>
Public Sub SayNextTest_1回目は1_1()


   Assert.AreEqual("1 - 1", result)
End Sub
```

テストケースのメソッド名は、それが何のテストであるか分かりやすく付けます。

Asset クラスで最も良く使うのは、この AreEqual() メソッドでしょう。 (最近の流行は、Asset.That() ですが、このチュートリアルでは使いません。)

Assert.AreEqual() の最初の引数は、期待値です。 こういう値になっているはず、という値を書きます。 ここでは "1 - 1" ですね。

2 番目の引数は、検査したい変数。 ここでは、返ってきた結果が result に入っているものとして書いています。

オプションで、3 番目の変数として、テストに失敗したときのメッセージを指定できます。

§ result を得るには?

この result は、どうやって得ることができるのでしたっけ? 外部設計のところを見ると、そうでした、FizzBuzzer クラスの SayNext() メソッドを呼び出せばよいのですね。

【最初のテストケース － 記述中(実行不可)】

```
<Test()>
Public Sub SayNextTest_1回目は1_1()


    Dim result As String = obj.SayNext()
    Assert.AreEqual("1 - 1", result)
End Sub
```

ローカル変数の宣言は、VB では Dim キーワードで始まり、変数名の後に As キーワードを付けて変数の型を書きます。 宣言だけでも良いですし、上のように、同時に初期化することもできます。

とりあえず FizzBuzzer クラスのインスタンス メソッドを呼び出すように書いてみました。 ここでちょっと考えてみます。 インスタンスを作った方が良いでしょうか、それとも、SayNext() メソッドはスタティック (VB では Shared キーワードを使います) の方が良いでしょうか? まぁ、迷ったらインスタンス メソッドでよいのですが、FizzBuzzer クラスは明らかに内部状態を持っていなければなりません (「番号」 を覚えていないと、次の 「番号」 が分かりません) ので、インスタンスを作らねばならないと分かります。

インスタンスを作るコードを加えて、テストケースは完成です。

【最初のテストケース】

```
<Test()>
Public Sub SayNextTest_1回目は1_1()
    Dim obj As FizzBuzzer = New FizzBuzzer()
    Dim result As String = obj.SayNext()
    Assert.AreEqual("1 - 1", result)
End Sub
```

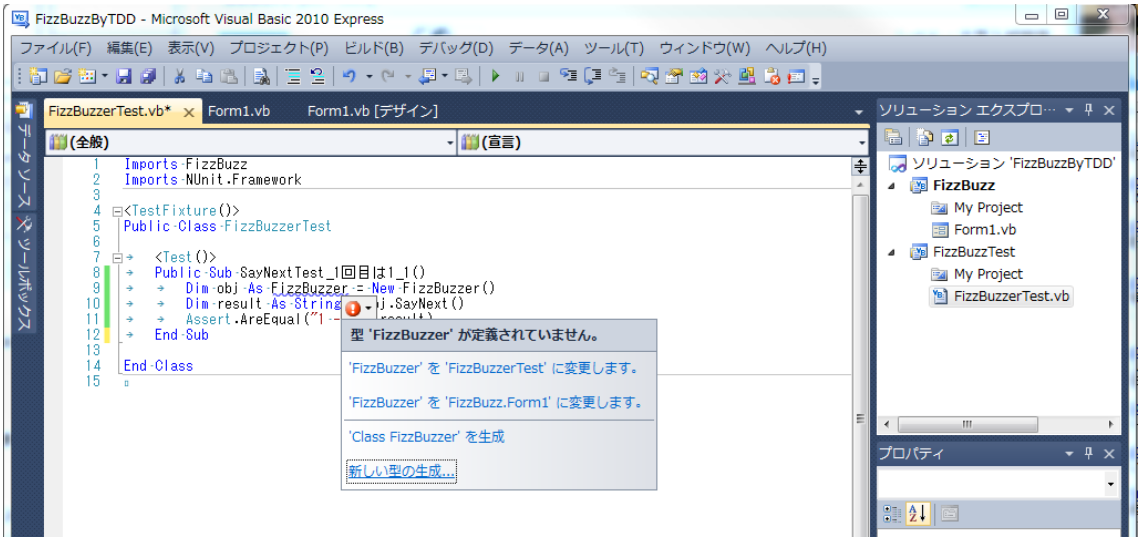
※ ちなみにこのあたりは、VB のインテリセンスが逆効果で、非常に書きづらいです。

§ 製品クラスを作る

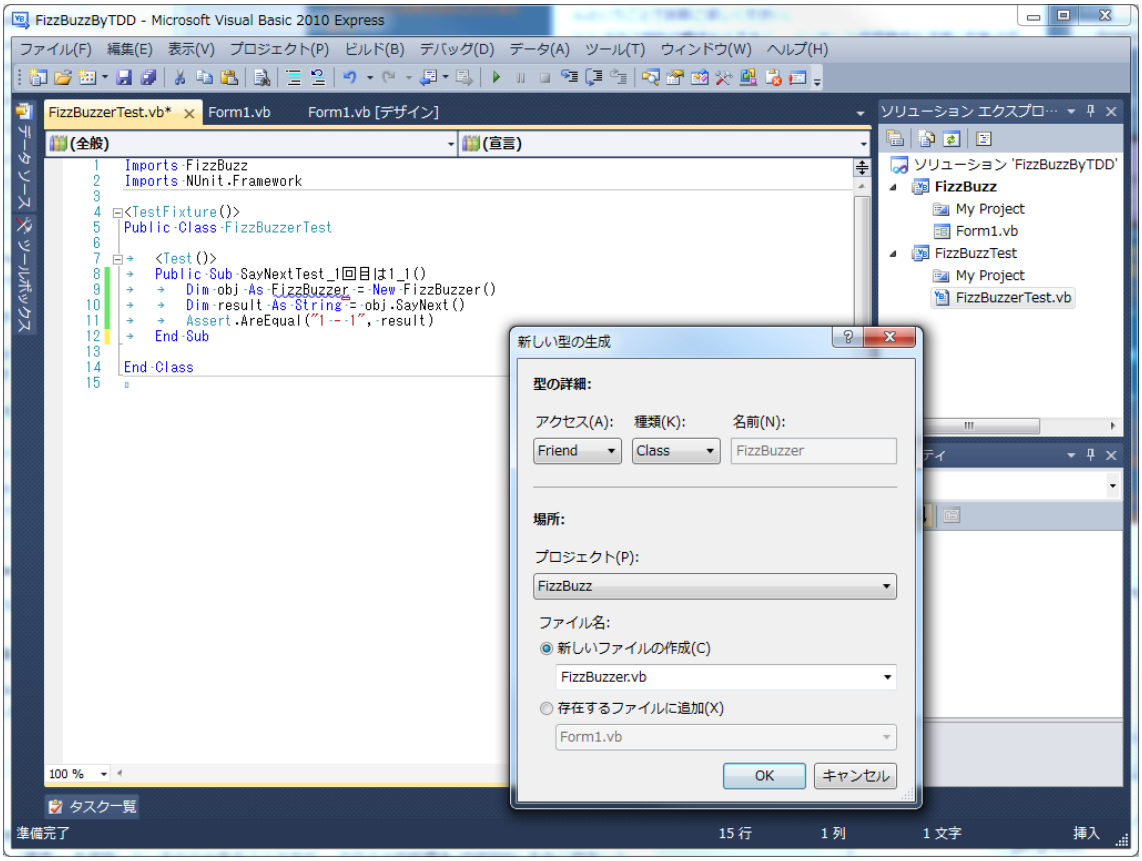
これで FizzBuzzTest プロジェクトをビルドしてみると、コンパイルエラーになります。作ってないから当たり前ですが、数人以上のチームで開発していると エラーにならない時があります (同じ名前のクラスを他人がすでに作っていたなど)。 エラーになるはずのことを、ちゃんと確かめて進むことも、大切です。

それでは、FizzBuzzer クラスを作りましょう。

コンパイルエラーになって下に波線が表示されている "FizzBusser" の所へ、マウスカーソルを持って行きます。 すると、赤いビックリマークが出てくるので、それをクリックします。



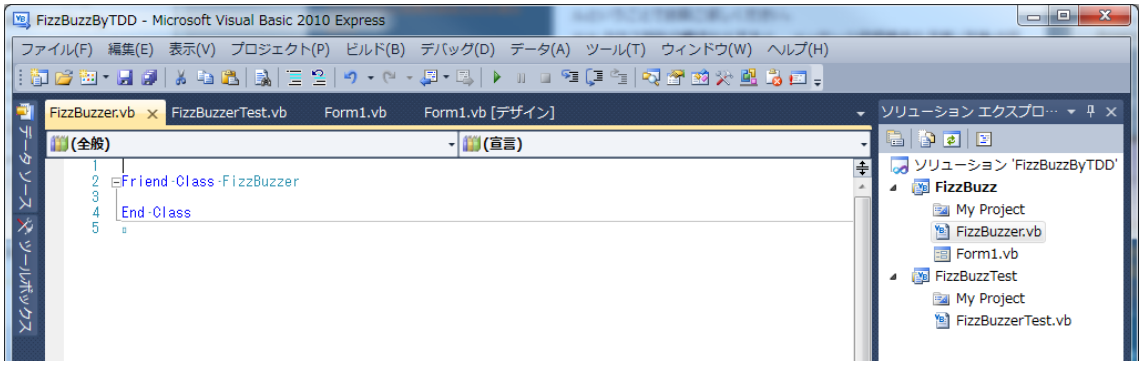
修正オプションが表示されるので、その中から [**新しい型の生成...**] を選びます。



[新しい型の生成] ダイアログは、下半分の 「場所」 を先に指定した方がいいです。 場所の指定を変えると、上の 「型の詳細」 がリセットされてしまうことがあり、かなりいらつくことになります。

新しいクラスを作る 「場所」 には、プロジェクトとして [FizzBuzz] を選び、ファイル名は、[新しいファイルの作成] を選んで "FizzBuzzer.vb" とします。 そして、「型の詳細」 ですが、これから作る SayNext() メソッドは Friend スコープですから、クラスのスコープも Friend にしておきます。 (上の画像を参照)

これで [OK] すると、次の画像のように、FizzBuzzer.vb ファイルが生成されます。



※ この画像は、自動生成後に手動で FizzBuzzer.vb ファイルを開いたところ。

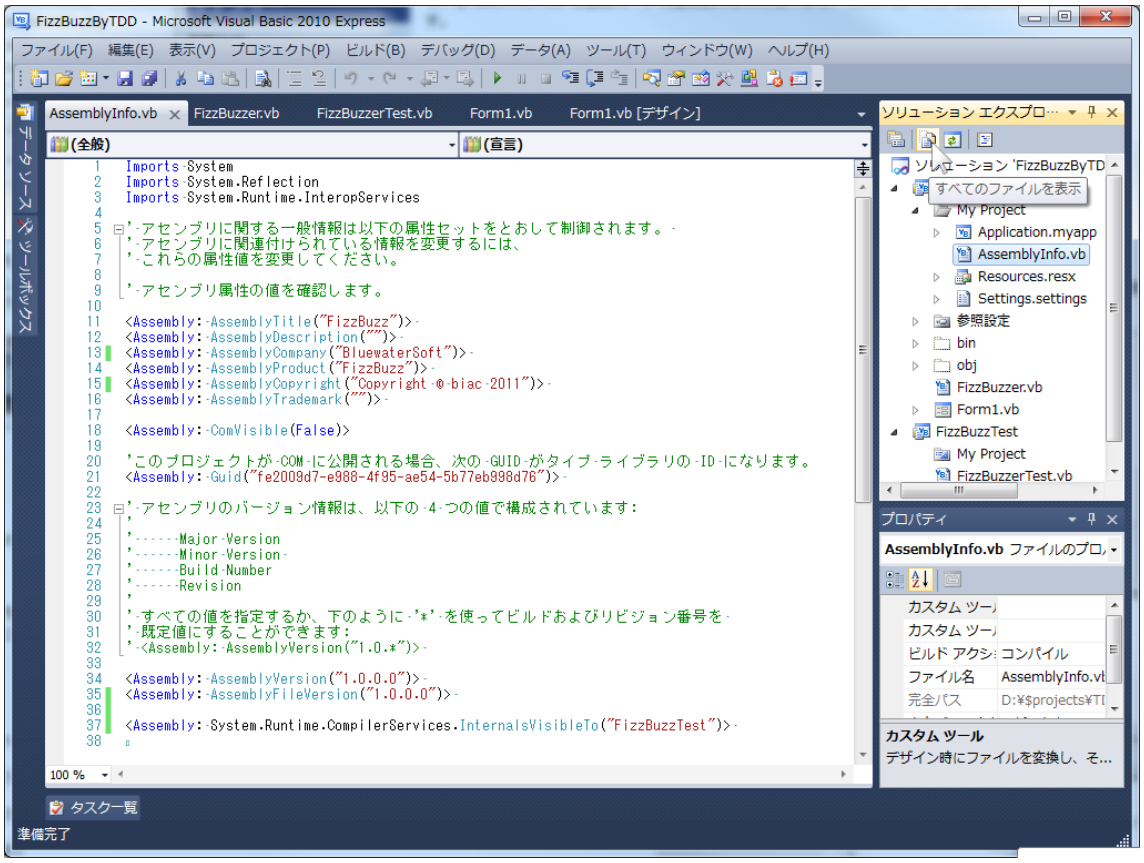
§ InternalsVisibleTo

FizzBuzzer.vb を作ったにも関わらず、まだコンパイルエラーです。 FizzBuzzer クラスを Friend スコープにしたので、FizzBuzz モジュールの外になる FizzBuzzTest からは「見えない」 のですね。

.NET Framework 2.0 からは、Friend スコープを指定したモジュールに公開するための InternalsVisibleTo 属性があります。 (**VB は 2008 から対応**)

この属性はモジュール全体に指定するのが一般的です。 そのためには **AssemblyInfo.vb** ファイルに記述します。ところが、デフォルトではソリューション エクスプローラーに AssemblyInfo.vb ファイルは表示されません。 ソリューション エクスプローラーの [**すべてのファイルを表示**] ボタンを使うと、FizzBuzz プロジェクトの下に My Project フォルダーが表示され、その中にあります。 ダブルクリックしてエディターで開いて、ファイルの末尾に次の 1 行を追加します。

```
<Assembly: System.Runtime.CompilerServices.InternalsVisibleTo("FizzBuzzTest")>
```



この指定により、FizzBuzz プロジェクト内の Friend スコープは、引数に指定した FizzBuzzTest プロジェクトにとっては、Public スコープと同じになります。

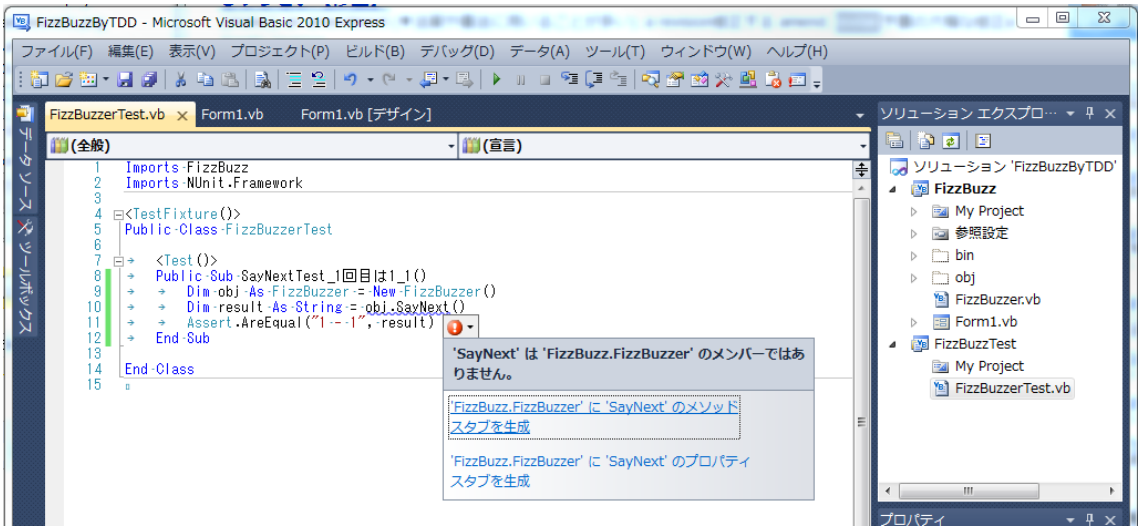
これで、FizzBuzzer をインスタンス化する行からは、エラーが消えました。

※ この行は、Friend スコープを公開したいモジュールごとに書く必要があります。

§ SayNext() の仮実装

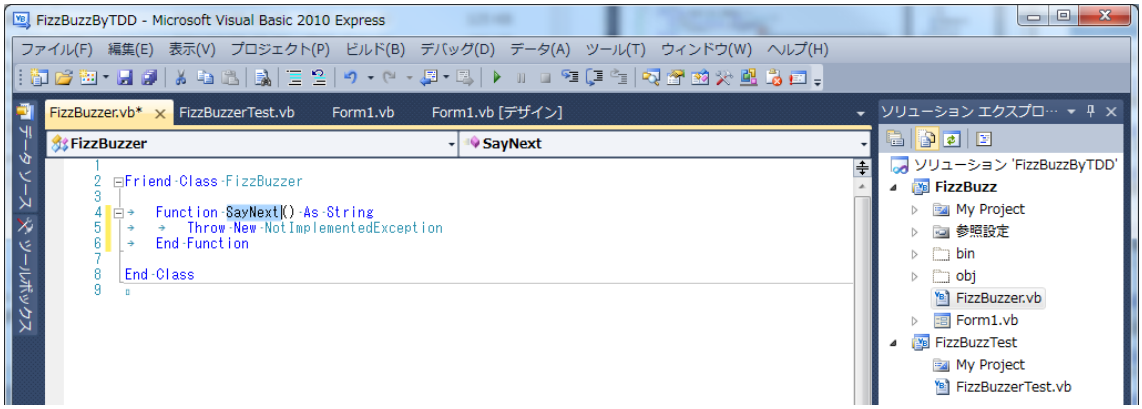
こんどは、obj.SayNext() がコンパイルエラーになっています。

このメソッドも、まだ作っていませんから、当然ですね。 メソッドを作って、とりあえずテストケースを通せるような、仮の実装をしましょう。 波線が表示されている obj.SayNext() の所にマウスカーソルを持って行くと、やはり赤いビックリマークが出ますから、それをクリックして出てくる修正候補の中から [**'FizzBuzz.FizzBuzzer' に 'SayNext' のメソッド スタブを生成**] を選びます。

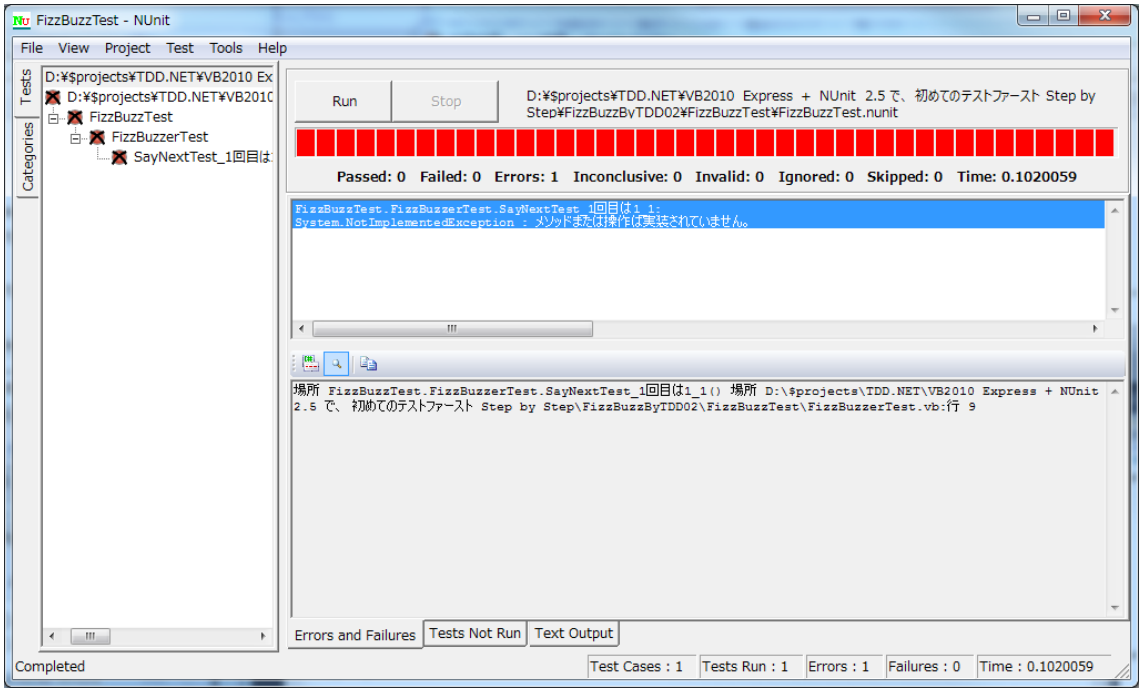


すると空っぽのメソッドが自動生成され、コンパイルエラーの表示は消えます。 先ほどクラスを自動生成したときもそうですが、自動生成後に画面が切り替わったりはせず、そのままコーディングを続けられるようになっています。

obj.SayNext() の SayNext を右クリックして [**定義へ移動**] を使うと、生成された SayNext() メソッドのコードが開きます。



自動生成されたメソッドは **NotImplementedException** 例外が投げられるようになっているので、このまま実行すると例外が出ます。 ビルドしてテストを実行し、そのことを確認してください。



このように、想定通りのエラーが出て失敗しましたでしょうか。

※ 64bit Windows を使っていて FizzBuzz が読み込めないというエラーが出た場合は、次項 「ターゲットプラットフォーム」 を参照してください。

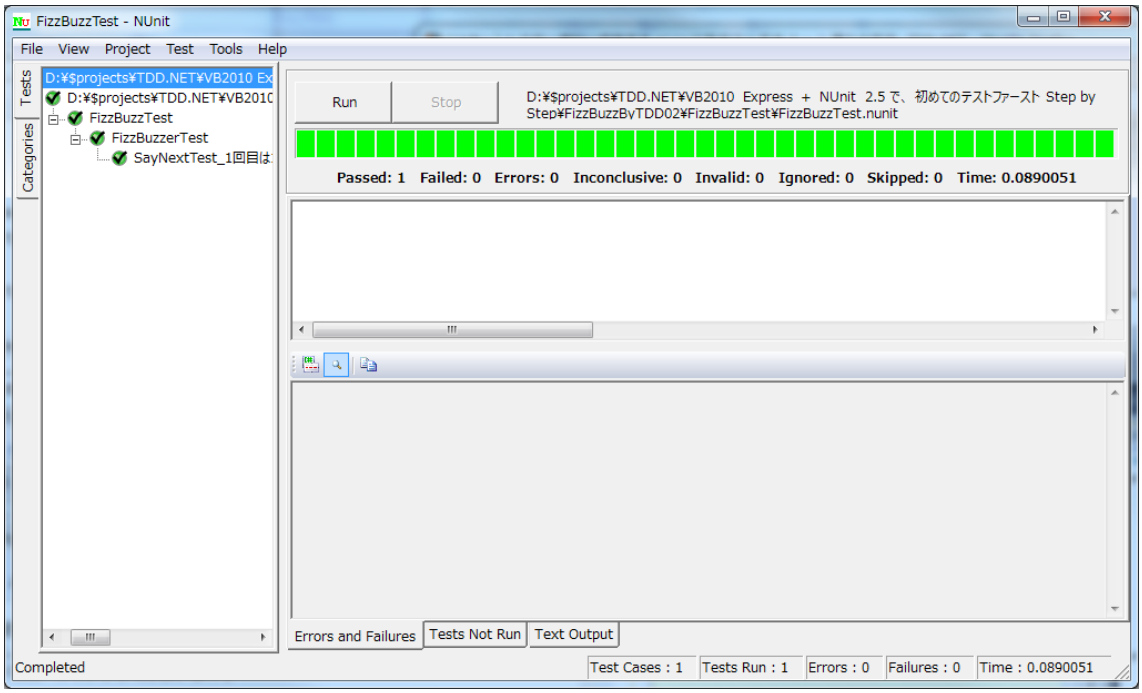
では、仮実装をしましょう。

いま通そうとしているテストケースは、**"1 - 1" という文字列を返しさえすれば OK** になります。 その通りにコーディングしてしまいます。

【製品クラス】

```
Friend Class FizzBuzzer


    Function SayNext() As String
        Return "1 - 1"
    End Function


End Class
```

ビルドして、NUnit でテストを実施します。 今度は通りますね。



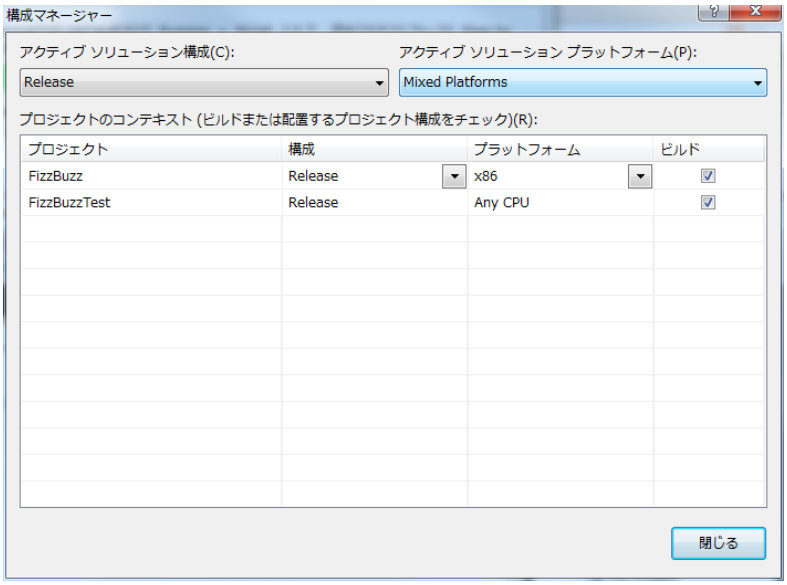
おめでとうございます! 初めてのグリーンです。

§ ターゲット プラットフォーム

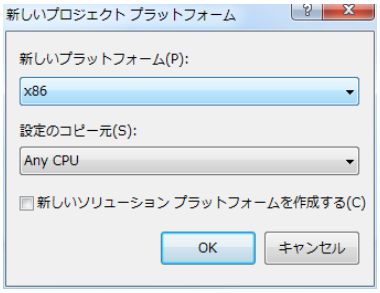
64bit の Windows では、テスト実行時に FizzBuzz を読み込めない、というエラーが NUnit から報告されることがあるかもしれません。 これは、FizzBuzz.exe と FizzBuzzTest.dll のターゲットプラットフォームが違ってしまっていて、片方は 32bit で動作しているのに、他方は 64bit で動いていることが考えられます。

ターゲット プラットフォームを合わせるには、次のようにします。

1. VB2010 のメニュー [ツール] - [オプション...] で、オプションダイアログを開き、[プロジェクトおよびソリューション] - [全般] で [ビルド構成の詳細を表示] にチェックを入れる。
2. メニュー [ビルド] - [構成マネージャー...] で 「構成マネージャー」 を開く。 画像のように、「アクティブ ソリューション プラットフォーム」 が [Mixed Platforms] になっている場合は、3. 以降の作業を行う。



3. 「アクティブ ソリューション プラットフォーム」 を [Any CPU] (32bit/64bit 兼用) か [x86] (32bit モードのみで動作) に変える。
4. プロジェクトのプラットフォームを、すべて 3. で指定したものに合わせる。 ドロップダウンに必要な選択肢が表示されていないときは、[<新規作成...>] を選んで作成する。 (表示される 「新しいプロジェクト プラットフォーム」 ダイアログで、[新しいソリューション プラットフォームを作成する] チェックボックスは外す)



☆ この時点でのソースコード一式 ⇒ 「FizzBuzzByTDD02.zip」 (25,671 バイト) をダウンロード http://www.tdd-net.jp/files/FizzBuzzByTDD02.zip (VB2010EE + NUnit2.5.9)

◆ 3-4: TDD 三原則

ふたつめのテストケースに取り掛かる前に、**TDD 三原則** (http://www.tdd-net.jp/2009/08/tdd-9534.html) を紹介しておきます。

正確には 「テストファースト三原則」 と言うべきもので、テストファーストするときには常に頭に入れておいてください。 実際の開発では、三原則から外れて作業することも

よくあります。 守らないことが悪いわけではなくて、そこはテストファーストで作らなかったと認識することが大事です。

> **TDD 三原則**: Robert C Martin (UncleBob) (翻訳 安井カ (やっとむ))
>
> 1. 失敗するユニットテストを成功させるためにしか、プロダクトコードを書いてはならない。
> 2. 失敗させるためにしか、ユニットテストを書いてはならない。 コンパイルエラーは失敗に数える。
> 3. ユニットテストを 1 つだけ成功させる以上に、プロダクトコードを書いてはならない。

さきほど、コンパイルエラーになると分かっていて先にテストケースのメソッド SayNextTest_1 回目は 1_1() を書きましたが、それは三原則の 2 番目です。 それから、SayNext() メソッドの実装を書いてテストに合格させたのは、1 番目の原則です。 その時、SayNext() メソッドの実装としては Return "1 - 1" だけではダメだと分かってはいますが、それ以上書くことは 3 番目の原則に従って我慢して、ダメなことを示す次のテストを書きます。

TDD 三原則を守ることによって、次のような効果が得られます。

- 製品コードはすべてユニットテストされている (原則 1, 3)
- ムダな製品コードを書かない (原則 1, 3)
- ムダなテストケースも書かない (原則 2)

なお、この原則に従ってテストファーストすると、メソッドの外部設計のところに記載した入出力表のすべてをテストコードに書くことにはなりません。 「これ以上は失敗するテストを書けない」 となった時点で、テストファーストは終了します。 多くの境界値はテストケースを書かないでしょうし、同値分割したケースを書かない場合も出てきます。

◆ 3-5: ふたつめのテストケース

§ レッド (テストケースを書いて失敗させる)

それでは、ふたつめのテストケースを作りましょう。 入出力表の、どれをやりましょうか?

- 失敗する (と考えられる) ケース (TDD 三原則の 2)
- なるべく簡単に製品コードを書けるケース (小刻みの法則)

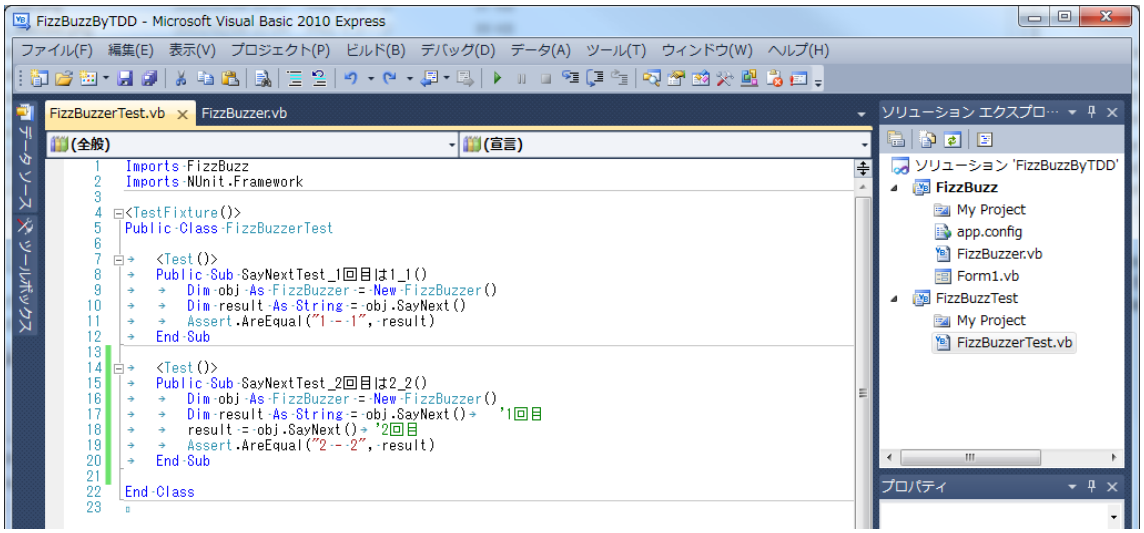
「小刻みの法則」 というのは、「易しいところから攻めろ!」 に置き換えてもいいです (少々意味は異なりますが)。 いきなり大きなステップを踏む (大量に製品コードを書く) というのは、バグった場合に何が原因かを突き止めるのが大変です。 遠回りのようでも、確実に一歩々々進みましょう。

「2 回目は "2 - 2"」 というケースは、失敗するはずですし、製品コードを書くのも簡単そうですね。 このテストケースを書きましょう。 FizzBuzzer のインスタンスを作ったら、SayNext() を 2 回呼び出せばよいのですね。

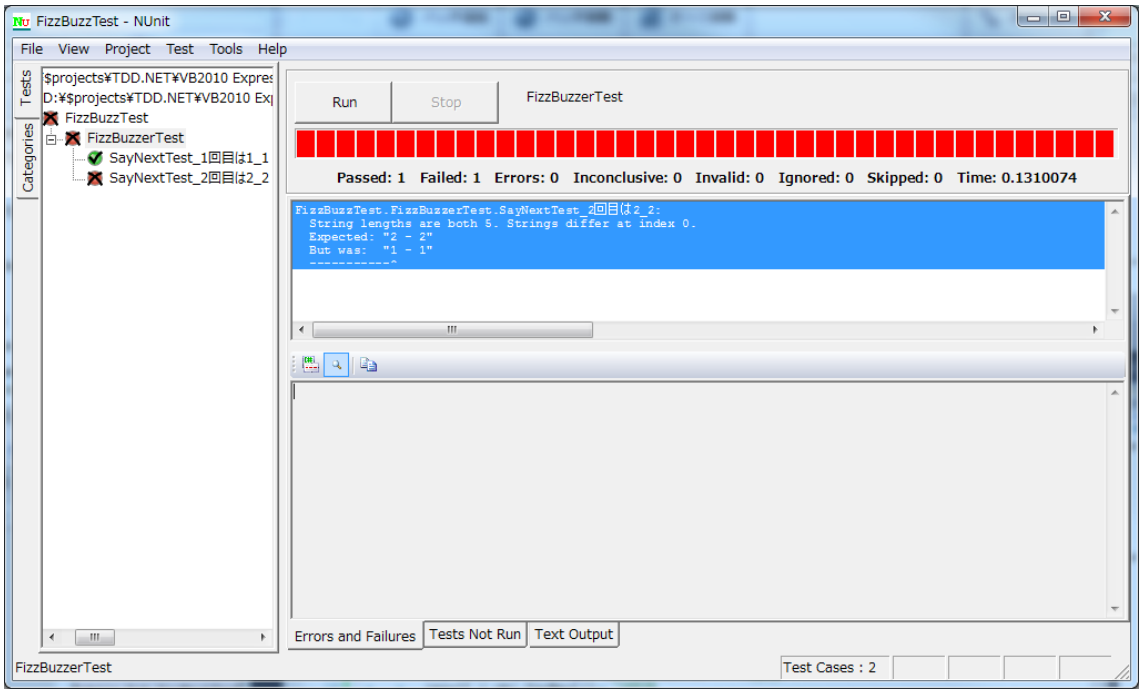
【ふたつ目のテストケースを追加】

```
<Test()>
Public Sub SayNextTest_1回目は1_1()
    Dim obj As FizzBuzzer = New FizzBuzzer()
    Dim result As String = obj.SayNext()
    Assert.AreEqual("1 - 1", result)
End Sub

<Test()>
Public Sub SayNextTest_2回目は2_2()
    Dim obj As FizzBuzzer = New FizzBuzzer()
    Dim result As String = obj.SayNext()    '1回目
    result = obj.SayNext()    '2回目
```

```
    Assert.AreEqual("2 - 2", result)
End Sub
```



ビルドして、NUnit でテストが失敗することを確認します。 期待値は "2 - 2" ですが、実際の返値は "1 - 1" のはずですね?



予想通りに失敗しましたか。 もしも、失敗はしたけどその内容が予想と違っていたならば、考え違いをしていたか、いま書いたテストコードにミスがあるかです。 その原因を解明しておきましょう。

§ グリーン (製品コードを書いて成功させる)

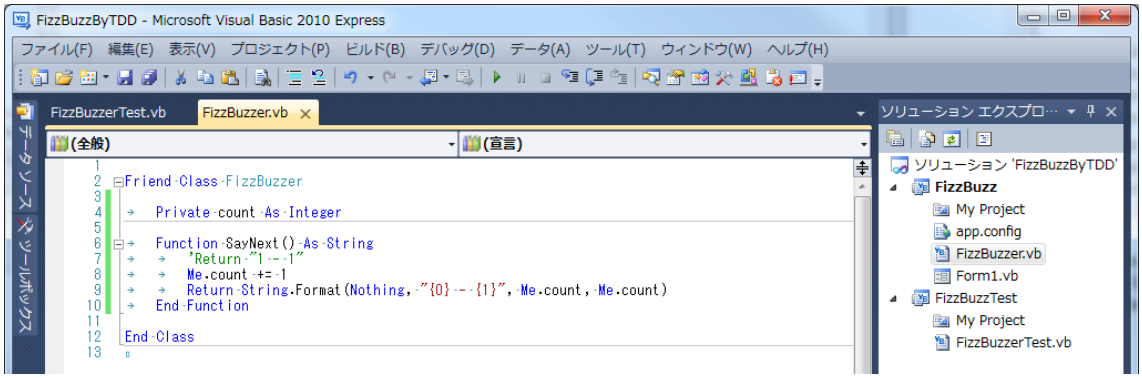
このテストを通すには、どうしたらよいでしょう?

- 何回目の呼び出しであるかを覚えておくメンバー変数が必要
- 回数 (数値) から返値の文字列 ("1 - 1", "2 - 2") を生成できる

この考え方で行けそうですね。 やってみましょう。

【製品クラス】

```
Friend Class FizzBuzzer

    Private count As Integer

    Function SayNext() As String
        'Return "1 - 1"
        Me.count += 1
        Return String.Format(Nothing, "{0} - {1}", Me.count, Me.count)
    End Function

End Class
```



変更前との比較のため、元のコードをコメントアウトして残してあります。 (実際の開発では、そんな面倒なことはしません。 どんどん書き換えていきます。)

テストファースト中は、製品コードの変数名などはいいかげんです。 一段落ついたところで、変数名などはリファクタリングします。

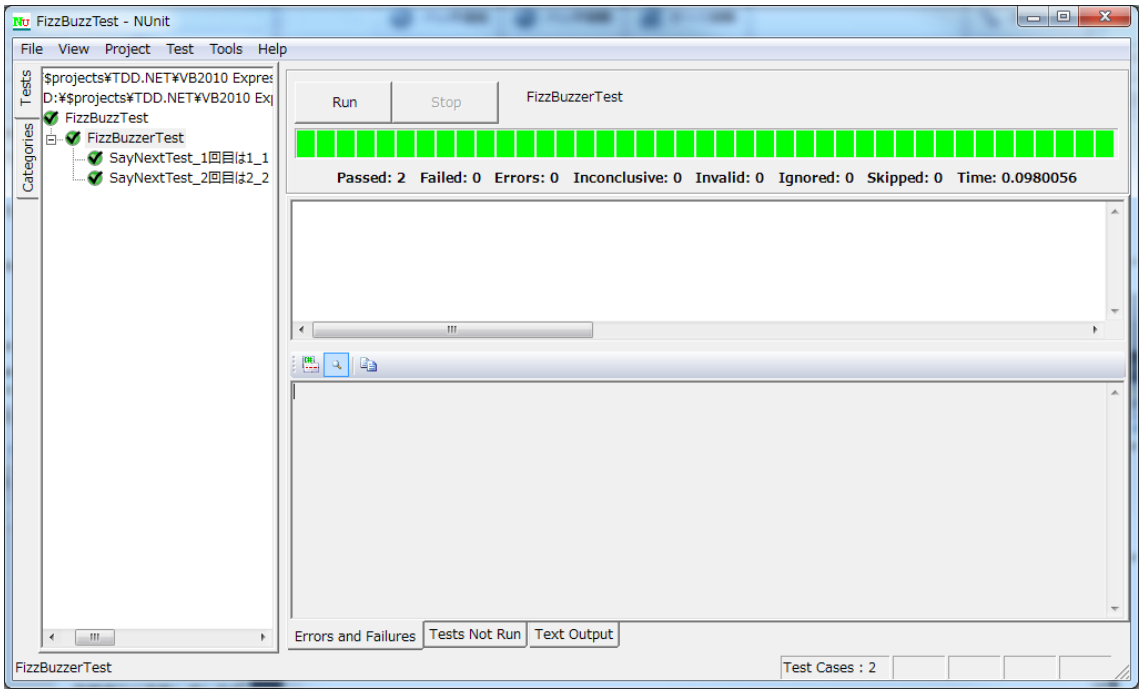
メンバー変数 count に初期値を与えていませんが、VB や C# ではメンバー変数は必ず 0 に初期化されます。

VB にインクリメント/デクリメント演算子 (++/--) は、ありません。 += や -= といった代入を伴う演算式はあります。

String クラスの Format() メソッドは、なんとなく分かるかと思います。 書式指定文字列中に変数を当てはめる場所を、中括弧「{}」で括った序数で示すところがちょっと風変わりかもしれません。 最初の引数の Nothing は null のことですが、その意味はとりあえず 「呪文」 だと思っておいてください。

※ VB の Nothing は null のようなものですが、値型にも代入出来て、そのときは 0 の意味になります。Java や C# より、VB の方が 「お約束」 が多くて難しいです。

ビルドして NUnit で確認します。



グリーン! ふたつめのテストケースも完了です。

◆ 3-6: テストケースのリファクタリング

みっつ目のテストケースに進む前に。

ここまでのふたつのテストケースを見てみると、かなり似ています。 次のテストケースも、同じようなコードになるでしょう。 似たようなテストをふたつかみっつ書いたら、テストコードのリファクタリングを考えるときです。

ふたつのテストケースの違いは、SayNext() を呼び出している回数と、Assert の期待値が異なるだけですね。 それぞれ少し書き直して、ひとつに出来るかどうか検討してみましょう。 なお、ここではていねいに解説していますが、多くの人はおそらく一気に書き直せると思います。

【ふたつのテストケースをそれぞれ修正】

```
<Test()>
Public Sub SayNextTest_1回目は1_1()
    Dim expected As String = "1 - 1"

    Dim obj As FizzBuzzer = New FizzBuzzer()
    Dim result As String '= obj.SayNext()
    result = obj.SayNext()     '1回目
    'Assert.AreEqual("1 - 1", result)
    Assert.AreEqual(expected, result)
End Sub

<Test()>
Public Sub SayNextTest_2回目は2_2()
    Dim expected As String = "2 - 2"

    Dim obj As FizzBuzzer = New FizzBuzzer()
    Dim result As String '= obj.SayNext()     '1回目
    result = obj.SayNext()     '1回目
    result = obj.SayNext()     '2回目
```

```
    'Assert.AreEqual("2 - 2", result)
    Assert.AreEqual(expected, result)
End Sub
```

書き換えたら、必ずビルドして NUnit でグリーンのままであることを確認します。

こうして書き直してみると、SayNext() を呼び出している行をループで回せば同じコードになりそうです。 ふたつめのテストケースを For ループで書き換えてみましょう。 このとき、いつでも引き返せるようにするため、既存のふたつのテストケースは消さずに残しておき、新しくテストケースを書きます。

【ふたつ目のテストケースの For ループ版】

```
<Test()>
Public Sub SayNextTest_2回目は2_2_Forループバージョン()
    Dim expected As String = "2 - 2"
    Dim count As Integer = 2

    Dim obj As FizzBuzzer = New FizzBuzzer()
    Dim result As String = Nothing
    For i As Integer = 1 To count
        result = obj.SayNext()
    Next
    Assert.AreEqual(expected, result)
End Sub
```

書き終えたらすぐにビルドして、NUnit でグリーンを確認します。

ループの書き方は、Basic の伝統が色濃く残っている部分です。 ループ変数をその場で宣言できるため、少々気持ち悪い形にはなっていますが、for ループは上に示したように記述します。 この場合、ループ変数 i は 1 から始まり count まで変化します (count - 1 までではありません)。

さて、この For ループ版、冒頭の expected と count に与える値を変えれば、ひとつめのテストケースにもなりますね。 NUnit では、Test 属性を TestCase 属性に変えて、次のようにまとめて書くことができます。

【TestCase 属性を使って、ふたつのテストケースをひとつのメソッドに】

```
<TestCase(1, "1 - 1")>
<TestCase(2, "2 - 2")>
Public Sub SayNextTest(ByVal count As Integer, ByVal expected As String)
    Dim obj As FizzBuzzer = New FizzBuzzer()
    Dim result As String = Nothing
    For i As Integer = 1 To count
        result = obj.SayNext()
    Next
    Assert.AreEqual(expected, result)
End Sub
```

TestCase 属性のひとつが、ひとつのテストケースになります。 それぞれのテストの際に、属性の後ろに指定した値が、テストメソッドの引数に渡されます。 たとえばひとつ目のテストでは、1 が count に、"1 - 1" が expected に与えられます。 これで次のテストケースからは、TestCase 属性を追加するだけで OK ですね。

なお、TestCase 属性が使えない場合は、次のようにして共通部分を括り出しておくとよいでしょう。

【TestCase 属性を使わずにリファクタリングした例】

```
Private Function CallSayNext(ByVal count As Integer) As String
    Dim obj As FizzBuzzer = New FizzBuzzer()
    Dim result As String = Nothing
    For i As Integer = 1 To count
        result = obj.SayNext()
    Next
```

```
    Return result
End Function

<Test()>
Public Sub SayNextTest_1回目は1_1_別案()
    Assert.AreEqual("1 - 1", Me.CallSayNext(1))
End Sub

<Test()>
Public Sub SayNextTest_2回目は2_2_別案()
    Assert.AreEqual("2 - 2", Me.CallSayNext(2))
End Sub
```

§ テストひとつに Assert ひとつの原則

一回のテストの実行で、ひとつのテストケースだけを行うべき、という原則。

ひとつのメソッドに複数のテストケースを詰め込んだのでは、読むときにテストケースの識別がしにくいだけでなく、NUnit がレッドになったときにもどのテストケースが失敗したのか分かりづらいし、ひいてはテストコードのメンテナンスも難しくなります。

ただし、言いやすいので 「Assert ひとつ」 としていますが、実際にはひとつのテストケースで複数の判定が必要なこともあります。 たとえば、製品コードを呼び出すと、返値だけでなく、データベースも書き換わる場合は、Assert 文がひとつでは足りないでしょう。

なお、TestCase 属性を使った場合は、ひとつのメソッドで複数のテストケースを兼ねていますが、これは先ほどのリファクタリングの例の後の方、TestCase 属性を使わない書き方の省略記法だと考えられます。

それでは、NUnit でグリーンになることを確認したら、残しておいた古いテストケースは消してしまって、次のテストケースに進みましょう。

◆ 3-7: レッド→グリーンのリズムに乗って

§ みっつめのテストケース

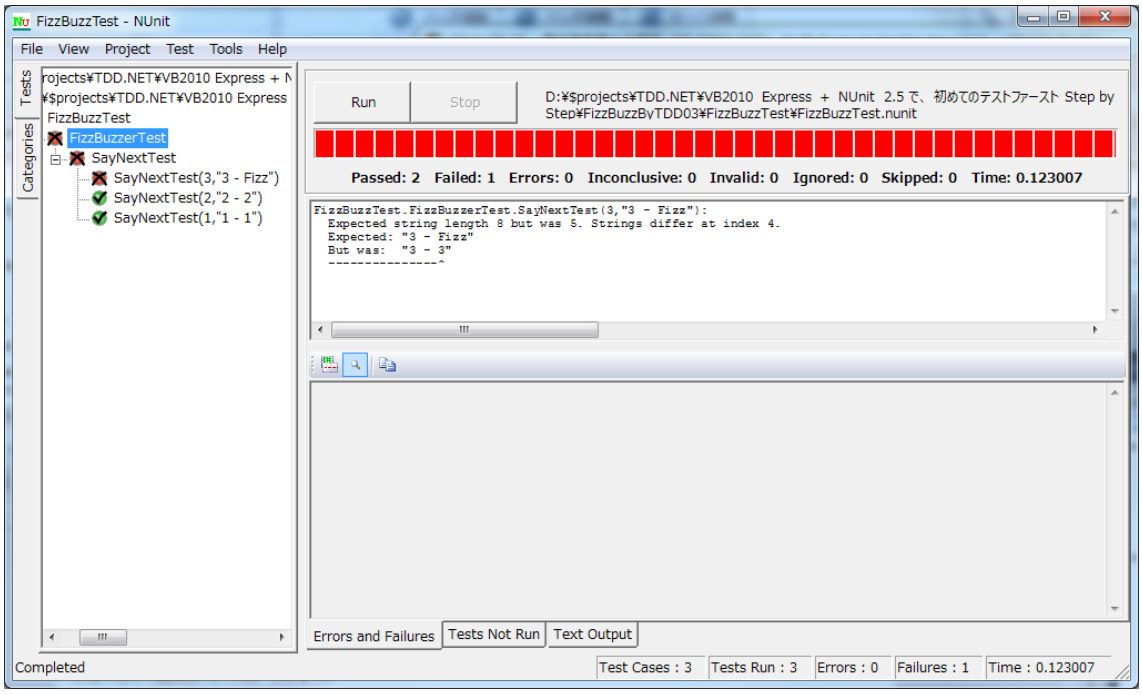
次はやはり 3 回目は "3 - Fizz" でしょう。

テストコードは、TestCase 属性を追加するだけです。

【みっつ目のテストケースを追加】

```
<TestCase(1, "1 - 1")>
<TestCase(2, "2 - 2")>
<TestCase(3, "3 - Fizz")>
Public Sub SayNextTest(ByVal count As Integer, ByVal expected As String)
    Dim obj As FizzBuzzer = New FizzBuzzer()
    Dim result As String = Nothing
    For i As Integer = 1 To count
        result = obj.SayNext()
    Next
    Assert.AreEqual(expected, result)
End Sub
```

ビルドして NUnit で予想通りに失敗することを確かめます。



このテストは、どうすれば通せるでしょう?

- String.Format() の最後の引数を、あらかじめ用意するように変える。
- 回数が 3 の倍数の時は、そこに回数ではなく "Fizz" を入れるようにする。

やってみましょう。

【製品クラス】

```
Friend Class FizzBuzzer

    Private count As Integer

    Function SayNext() As String
        Me.count += 1

        Dim s As String = Me.count.ToString()
        If ((Me.count Mod 3) = 0) Then
            s = "Fizz"
        End If

        'Return String.Format(Nothing, "{0} - {1}", Me.count, Me.count)
        Return String.Format(Nothing, "{0} - {1}", Me.count, s)
    End Function

End Class
```

整数を文字列に変換するには、このように ToString() メソッドを使います。 ここではやっていませんが、変換時のフォーマットを指定することもできます。

VB で剰余を求める演算子は Mod です。

If 文の中で、等しいことを判定するには = を 1 個です (ということは、If 文の中で代入は出来ません)。

ビルドして、NUnit でグリーンを確認します。

§ ４番目のテストケース

次は、5 回目には "5 - Buzz" が順当なところでしょう。

【4番目のテストケースを追加】

```
<TestCase(1, "1 - 1")>
<TestCase(2, "2 - 2")>
<TestCase(3, "3 - Fizz")>
<TestCase(5, "5 - Buzz")>
Public Sub SayNextTest(ByVal count As Integer, ByVal expected As String)
    (…以下略)
```

ビルドして、NUnit が予想通りのレッドになることを確かめます。
製品コードもすぐに直せます。

【製品コード】

```
    Function SayNext() As String
        Me.count += 1

        Dim s As String = Me.count.ToString()
        If ((Me.count Mod 3) = 0) Then
            s = "Fizz"
        ElseIf ((Me.count Mod 5) = 0) Then
            s = "Buzz"
        End If

        Return String.Format(Nothing, "{0} - {1}", Me.count, s)
    End Function
```

ビルドして NUnit がグリーンになることを確認します。

§ 5 番目のテストケース

次は、15 回目に "15 - Fizz Buzz" でしょう。

【5番目のテストケースを追加】

```
<TestCase(1, "1 - 1")>
<TestCase(2, "2 - 2")>
<TestCase(3, "3 - Fizz")>
<TestCase(5, "5 - Buzz")>
<TestCase(15, "15 - Fizz Buzz")>
Public Sub SayNextTest(ByVal count As Integer, ByVal expected As String)
    (…以下略)
```

ビルドして、NUnit がレッドになることを確認します。 予想通りに、"15 - Fizz" が返ってきて失敗しましたか?

製品コードのほうは、どのように修正しましょうか。

"Fizz Buzz" と答えるべきは、3 の倍数 かつ 5 の倍数の時です。 既存の If 文の後ろに ElseIf を付けてその判定をしたとすると、上手くいくでしょうか? たぶんダメですね、そこに来るときは 3 の倍数でもなく 5 の倍数でもないときですから。

既存の If 文の前だったら…? いけそうですね、やってみましょう。

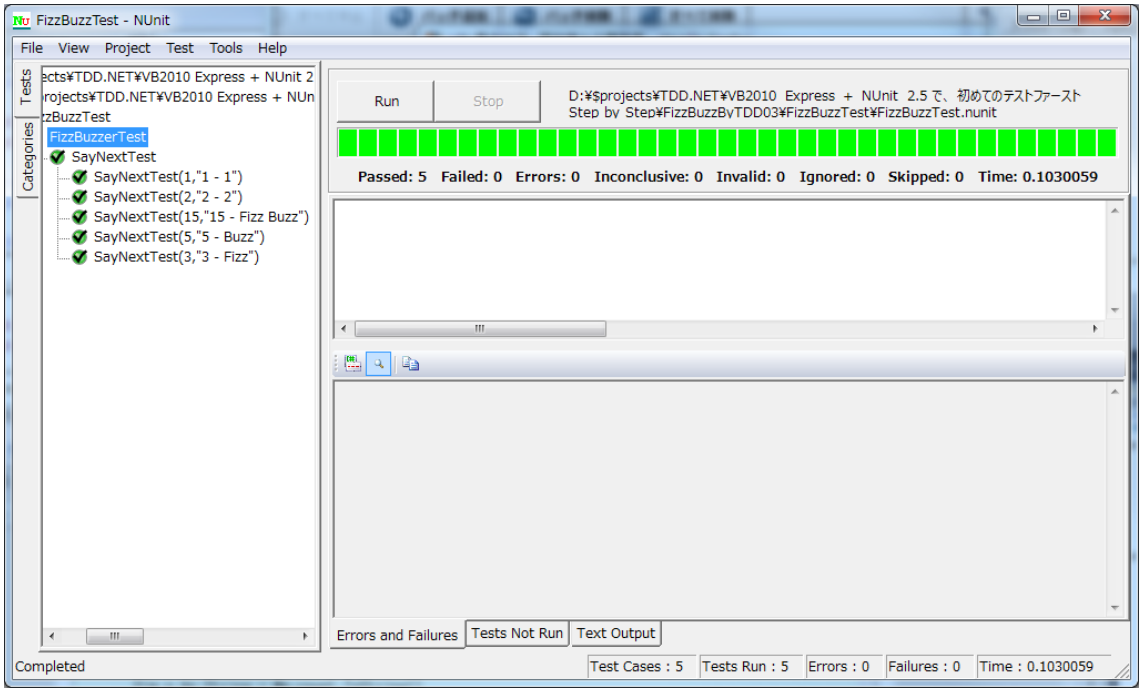
【製品コード】

```
Function SayNext() As String
    Me.count += 1

    Dim s As String = Me.count.ToString()
    If ((Me.count Mod 3) = 0 AndAlso (Me.count Mod 5) = 0) Then
        s = "Fizz Buzz"
    ElseIf ((Me.count Mod 3) = 0) Then
        s = "Fizz"
    ElseIf ((Me.count Mod 5) = 0) Then
        s = "Buzz"
    End If
```

```
        Return String.Format(Nothing, "{0} - {1}", Me.count, s)
    End Function
```

VB の条件式での AND と OR は、AndAlso と OrElse であると覚えてください。(And と Or もありますが、挙動が違い、ショートサーキットしません。)

それではビルドして、NUnit からグリーンの祝福をもらいましょう!



◆ 3-8: テストファースト終了

ここまで書いてきたテストケースの他に、失敗させることが出来るテストケースはあるでしょうか?

- 同値分割して見出した 4 つの同値クラスは、すべてテストされています。
- 同値クラス間の境界値をテストしていないところが多いですが、製品コードには不等号を使った比較は無く、どこにも失敗しそうなところは見当たりません。
- Integer.MaxValue 回目のケースも、失敗させられそうもありません。
- (Integer.MaxValue + 1) 回目のケースは、実行すれば例外が出るのは明らかで、要求を満たしています。

もう思い付けなかったら…

テストファースト完了です♪

☆ この時点でのソースコード一式 ⇒ 「FizzBuzzByTDD03.zip」 (26,406 バイト) をダウンロード http://www.tdd-net.jp/files/FizzBuzzByTDD03.zip (VB2010EE + NUnit2.5.9)

■ 4. TDD (リファクタリング)

「3-1: メソッドの外部設計」 で決めた通りの SayNext() メソッドが出来上がりました。さっそく UI と繋げてみたくなりますが、記憶が新たなうちに SayNext() メソッドのリファクタリングを先に実施します。

◆ 4-1: 外部設計と内部設計について

メソッドの外部設計は、そのメソッドを利用する側にしてみれば、いわば契約です。 コロコロ変えられたのではたまったものではありません。 それに対して、内部設計はそれを作る人が好き勝手にしても、利用する側には影響がありません。

TDD では、**テストファーストで外部設計をまず固定**し、それから**内部設計を改善するために 「リファクタリング」** を行います。 内部設計は凝りだしたら際限がありませんので、とりあえずは 「あとでコードを読んだときに理解できるだろうか?」 というのを最低の基準とし、それ以上は、実際に改修することになったときや時間に余裕ができたときにリファクタリングを行うようにするとよいでしょう。

◆ 4-2: リファクタリングとは

リファクタリングについて学ぶには、いまでもこの本が最良だと思います。

「**リファクタリング ― プログラムの体質改善テクニック**」
マーチン ファウラー (著)
ピアソンエデュケーション (2000/05)
ISBN-13: 978-4894712287
※ VB のコードしか読めないようでは、この本を読むのは辛いですが、C# のコードが読めるなら大丈夫です。

マーチン ファウラーによれば、リファクタリングは次のように定義されます。

リファクタリング (名詞)

外部から見たときの振る舞いを保ちつつ、理解や修正が簡単になるように、ソフトウェアの内部構造を変更させること。

リファクタリング (動詞)

一連のリファクタリングを行って、外部から見た振る舞いの変更なしに、ソフトウェアを再構築すること。

「外部から見たときの振る舞いを保ちつつ」ということは、外部設計を変えずに、あるいは、利用者への契約を守ったままにする、ということですね。 それを保証できない作業は、ファウラーの定義に従えばリファクタリングではないということになります。 保証するにはどうすればよいかと言えば、内部構造を変える前と後で振る舞いが変わっていないことを証明すればよいわけで、それは自動化されたテストがあれば簡単です。

ここまでテストファーストで作ってきた部分には、自動化されたユニットテストも出来上がっていますから、憂うことなくリファクタリングを行うことが出来ます。

ひとつ注意しておくと、「外部から見たときの振る舞いを保ちつつ」 ということには、新しい振る舞いも追加しない、という意味も含まれています。 新しい振る舞いが必要となったら、TDD 三原則を思い出して、テストファーストしてください。

◆ 4-3: リファクタリングする候補を挙げる

リファクタリングは、製品コードをじっくり眺めてみるところから始まります。 SayNext() メソッドにコメントを付けて再掲します。 分かりにくいところや、気に食わないところはどこですか?

【製品コード】

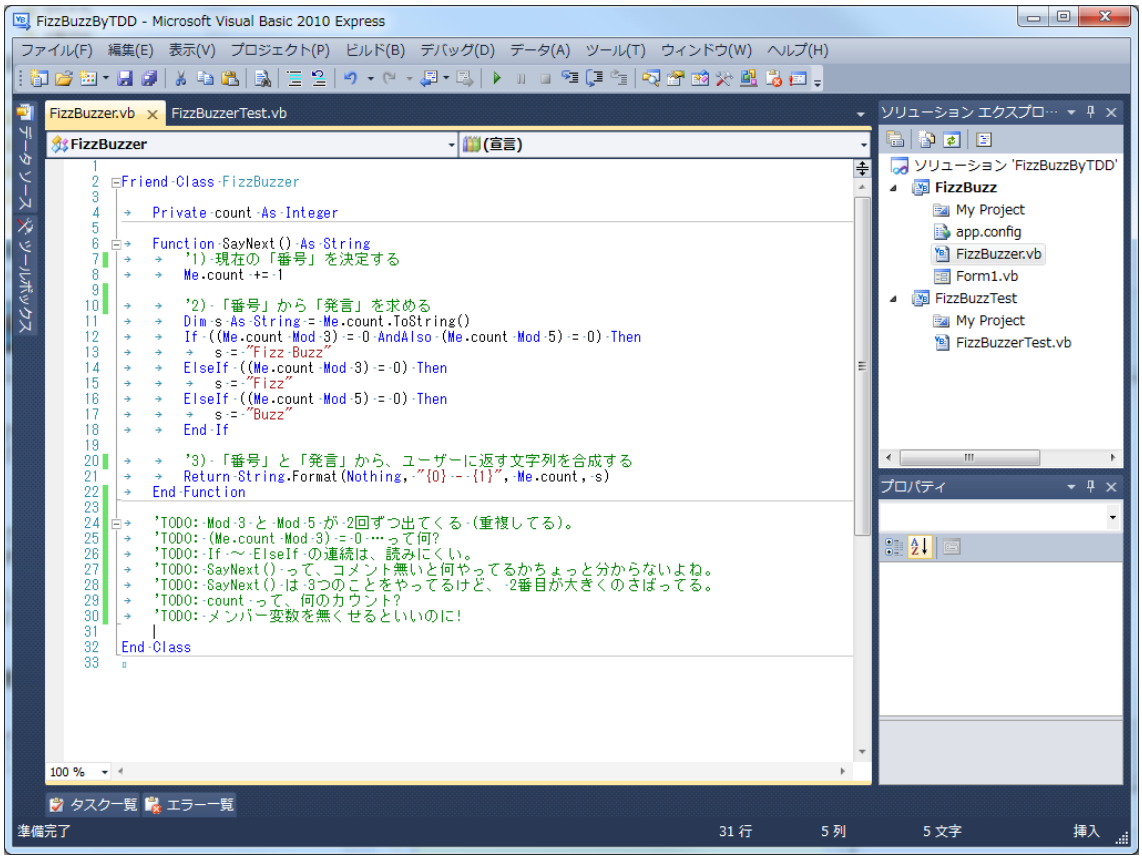
```
Friend Class FizzBuzzer

    Private count As Integer

    Function SayNext() As String
```

```
        '1) 現在の「番号」を決定する
        Me.count += 1

        '2) 「番号」から「発言」を求める
        Dim s As String = Me.count.ToString()
        If ((Me.count Mod 3) = 0 AndAlso (Me.count Mod 5) = 0) Then
            s = "Fizz Buzz"
        ElseIf ((Me.count Mod 3) = 0) Then
            s = "Fizz"
        ElseIf ((Me.count Mod 5) = 0) Then
            s = "Buzz"
        End If

        '3) 「番号」と「発言」から、ユーザーに返す文字列を合成する
        Return String.Format(Nothing, "{0} - {1}", Me.count, s)
    End Function
End Class
```

追加したコメントのうち、3) は無くても分かるでしょう。 1) と 2) のコメントは、あって欲しいのではないでしょうか? コメントが欲しいという所も、リファクタリングする候補になります。

そのほか、思いついたものを書きだしてみましょう。 (実際の開発では、その場でやれるものはすぐにリファクタリングしてしまって、後でやりたいものだけメモとして残しておきます。)

- Mod 3 と Mod 5 が 2 回ずつ出てくる (重複してる)。
- (Me.count Mod 3) = 0 …って何?
- If ～ ElseIf の連続は、読みにくい。
- SayNext() って、コメント無いと何やってるかちょっと分からないよね。
- SayNext() は 3 つのことをやってるけど、2 番目が大きくのさばってるなぁ。
- count って、何のカウント?
- メンバー変数を無くせるといいのに!



§ 日本語の名前

そうそう、分かりやすいと言えば、変数名やメソッド名やクラス名。 Visual Studio で変数名などに日本語を使うことには、もはやほとんど不安は無くなりました。 プログラムの内部だけで使う分には、コード解析ツールなどで文字化けすることが問題になるくらいです。

外部に公開するもの (他プログラムからの利用を前提としたライブラリー等) や、外部の名前を取り込む場合 (データベースのフィールド名など) には、文字コードの変換が介在する可能性が高いので、日本語の利用はまだ避けるべきでしょう。

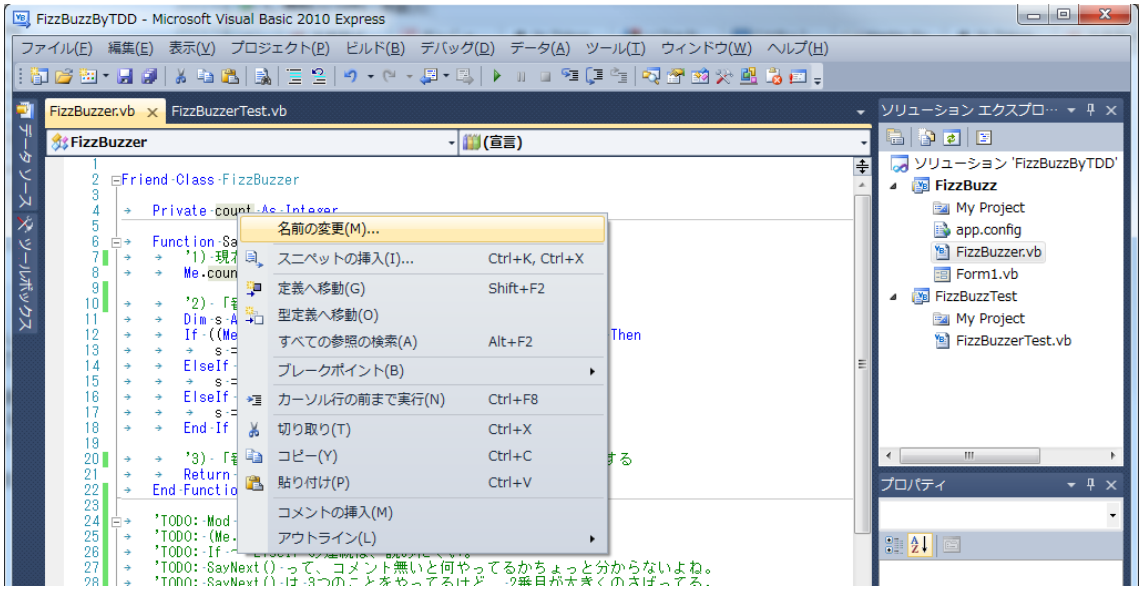
ここまでも、テストケースにはメソッド名に日本語を使ってきました。 製品コードにも使うことにしましょう。

◆ 4-4: リファクタリングを行う

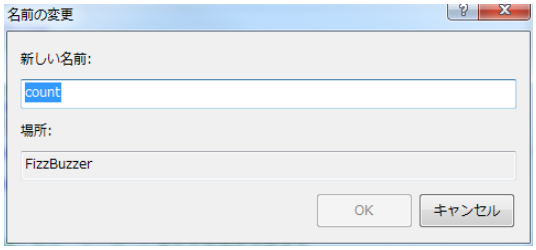
§ 変数のリネーム (Rename)

リファクタリングのカタログに載っていないほどに、ありふれたリファクタリングです。

メンバー変数 count は、何のカウントだか分かりませんね。 callCount (呼び出しカウント) に直しておきましょう。



リネームしたい変数 (ここでは "count") を右クリックして [名前の変更...] を選びます。



出てきた 「名前の変更」 ダイアログで、新しい変数名 (ここでは "callCount") に書き換えて、[OK] します。 これで、その変数を参照している箇所も、すべて変更されます。

NUnit で関連しそうな (ここでは、全ての) テストケースを実行して、グリーンのままであることを確認します。

§ メソッドへ切り出し (Extract Method)

メソッドの一部分を新しいメソッドとして切り出すことで、元のメソッドは短くなって見通しが良くなるとともに、切り出した部分に名前が付くことによって理解しやすくなります。

※ ここでは手動で行います。 Visual Studio の上位バージョンでは、自動的に行えます。

まず、「2) 「番号」から「発言」を求める」 とコメントした部分を切り出しましょう。新しく作るメソッドの名前はどうしましょうか? "Get 発言 From 番号()" というような感じにしてもよいですが、番号を引数で渡すことにすれば "Get 発言(番号)" となって短くなりますね。 後者でいきましょう。

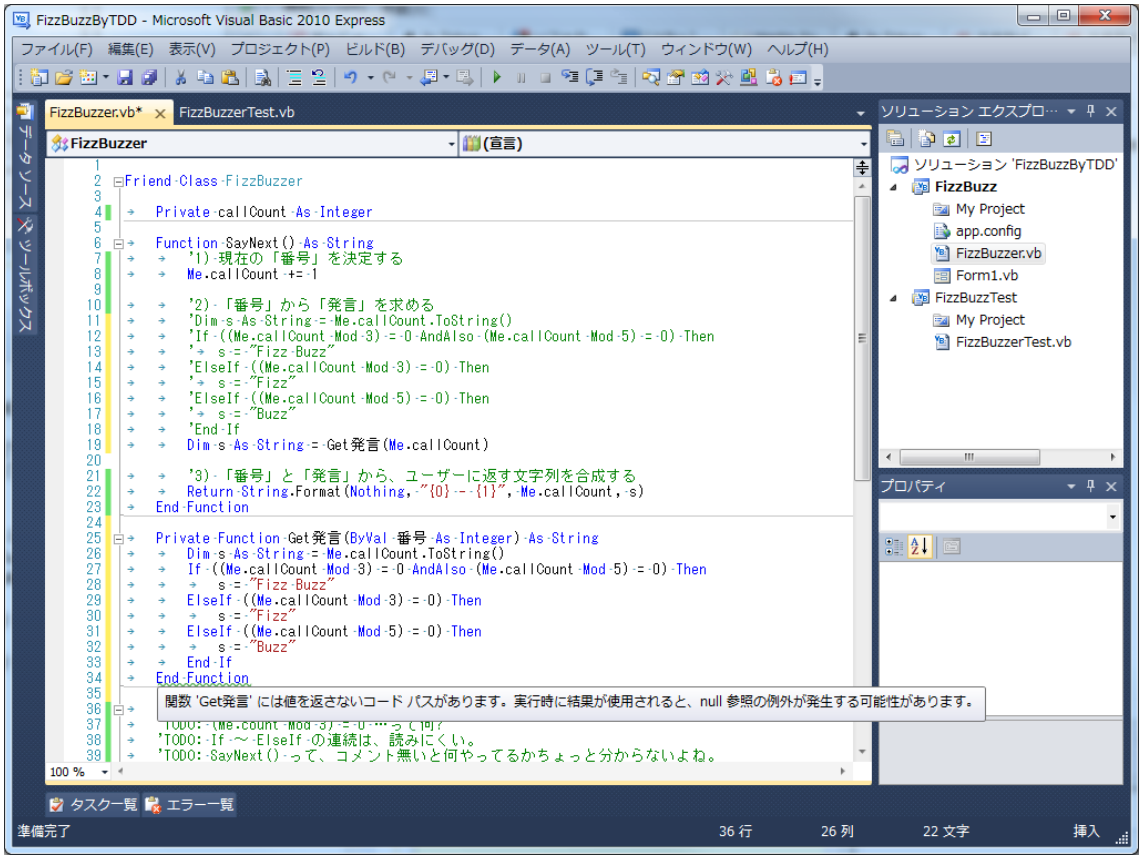
最初に、新しく作るメソッドの 「枠」 を書きます。

```
    Private Function Get発言(ByVal 番号 As Integer) As String


    End Function
```

次に、切り出したいコードをコピー&ペーストして、元のコードは (失敗したらすぐ戻せるように) コメントアウトし、新しいメソッドへの呼び出しを追加します。

```
    Function SayNext() As String
        '1) 現在の「番号」を決定する
        Me.callCount += 1


        '2) 「番号」から「発言」を求める
        'Dim s As String = Me.callCount.ToString()
        'If ((Me.callCount Mod 3) = 0 AndAlso (Me.callCount Mod 5) = 0) Then
        '    s = "Fizz Buzz"
        'ElseIf ((Me.callCount Mod 3) = 0) Then
        '    s = "Fizz"
        'ElseIf ((Me.callCount Mod 5) = 0) Then
        '    s = "Buzz"
        'End If
        Dim s As String = Get発言(Me.callCount)
```

```
        '3) 「番号」と「発言」から、ユーザーに返す文字列を合成する
        Return String.Format(Nothing, "{0} - {1}", Me.callCount, s)
    End Function

    Private Function Get発言(ByVal 番号 As Integer) As String
        Dim s As String = Me.callCount.ToString()
        If ((Me.callCount Mod 3) = 0 AndAlso (Me.callCount Mod 5) = 0) Then
            s = "Fizz Buzz"
        ElseIf ((Me.callCount Mod 3) = 0) Then
            s = "Fizz"
        ElseIf ((Me.callCount Mod 5) = 0) Then
            s = "Buzz"
        End If
    End Function
```



コピー&ペーストした新しいメソッド Get 発言() の中を修正していきます。 Return 文を付け加え、Me.callCount を引数の番号に書き換えればよいですね。 (ここで 「名前の変更」 機能を使うと、メンバー変数全体が変わってしまいます。 手動でリネームしなければなりません。)

```
    Private Function Get発言(ByVal 番号 As Integer) As String
        Dim s As String = 番号.ToString()
        If ((番号 Mod 3) = 0 AndAlso (番号 Mod 5) = 0) Then
            s = "Fizz Buzz"
        ElseIf ((番号 Mod 3) = 0) Then
            s = "Fizz"
        ElseIf ((番号 Mod 5) = 0) Then
            s = "Buzz"
        End If
        Return s
    End Function
```

あれ? このメソッドはメンバー変数に依存しなくなりました。 スタティックメソッドにしちゃっても OK ですね。 VB では static ではなく、Shared キーワードを使います。

```
    Private Shared Function Get発言(ByVal 番号 As Integer) As String
        Dim s As String = 番号.ToString()
        If ((番号 Mod 3) = 0 AndAlso (番号 Mod 5) = 0) Then
            s = "Fizz Buzz"
        ElseIf ((番号 Mod 3) = 0) Then
            s = "Fizz"
        ElseIf ((番号 Mod 5) = 0) Then
            s = "Buzz"
        End If
        Return s
    End Function
```

ビルドして、間違えていないことを NUnit にグリーンで教えてもらえたら、コメントアウトして残しておいた元のコードを削除します。

§ 説明用の変数の導入 (Introduce Explaining Variable)

SayNext() メソッドは、現在こんなふうになっています。

```
    Function SayNext() As String
        '1) 現在の「番号」を決定する
        Me.callCount += 1

        '2) 「番号」から「発言」を求める
        Dim s As String = Get発言(Me.callCount)

        '3) 「番号」と「発言」から、ユーザーに返す文字列を合成する
        Return String.Format(Nothing, "{0} - {1}", Me.callCount, s)
    End Function
```

短くはなりましたが、コードの 1 行ごとにコメントが付いていてうっとおしいです。まず、3 番目のコメントは無くても分かりますね。 削ってしまいましょう。 1 番目と 2 番目のコメントはどうでしょう? "番号"という変数を導入すると分かりやすくなって、コメントは不要になります。

```
    Function SayNext() As String
        Me.callCount += 1

        Dim 番号 As Integer = Me.callCount
        Dim s As String = Get発言(番号)

        Return String.Format(Nothing, "{0} - {1}", 番号, s)
    End Function
```

こうしてみると、1 行目は現在の「番号」を決定していたのではなくて、単に呼び出しカウントをインクリメントしているだけだということが、改めて認識できますね。 そして、番号が呼び出しカウントに等しい値であると定義されていることも。

それではビルドして、外部設計に変化が無いことを、NUnit でテストを実施して確認しておきます。

§ ふたたび変数のリネーム (Rename)

SayNext() メソッド内の変数 s は、すぐ次の行で使って終わりですから、そのままでも構わないのですが、"発言" にリネームしておくと分かりやすくなります。

```
    Function SayNext() As String
        Me.callCount += 1

        Dim 番号 As Integer = Me.callCount
        Dim 発言 As String = Get発言(番号)

        Return String.Format(Nothing, "{0} - {1}", 番号, 発言)
    End Function
```

このくらいの小変更なら、壊していない自信は十分以上にあるでしょう。 テストはサボってしまいましょうか。

§ ふたたび説明用の変数の導入 (Introduce Explaining Variable)

次は、切り出した Get 発言() メソッドをリファクタリングしていきましょう。

If 文がゴチャゴチャしていて理解しづらいのですが、まず論理式の部分をすっきりさせましょうか。 論理式の部分をブール型の変数に代入してから使うようにします。

```
    Private Shared Function Get発言(ByVal 番号 As Integer) As String
        Dim is3の倍数 As Boolean = ((番号 Mod 3) = 0)
        Dim is5の倍数 As Boolean = ((番号 Mod 5) = 0)

```

```
        Dim s As String = 番号.ToString()
        If (is3の倍数 AndAlso is5の倍数) Then
            s = "Fizz Buzz"
        ElseIf (is3の倍数) Then
            s = "Fizz"
        ElseIf (is5の倍数) Then
            s = "Buzz"
        End If
        Return s
    End Function
```

このリファクタリングでは、以前は 3 と 5 で 2 回ずつあった剰余演算が 1 回ずつに減るというオマケも付きました。 さらに "is15 の倍数" を導入しても構いません。

それでは、ビルドしてグリーンになることを確認しておいてください。

§ ネストした条件式のガード句による置き換え (Replace Nested Conditional with Guard Clauses)

入れ子になった If 文や ElseIf の連続など複雑な条件式は、ガード句の形にするときれいにできる場合があります。 ガード句というのは、メソッドの先頭の方で引数をチェックし、ダメだったらただちにリターンさせてしまう書き方です。 ここでは、返すべき値が決まったら、その時点でリターンさせます。 使い道が異なっていても、ファウラーはガード句と呼んでいるようです。

```
    Private Shared Function Get発言(ByVal 番号 As Integer) As String
        Dim is3の倍数 As Boolean = ((番号 Mod 3) = 0)
        Dim is5の倍数 As Boolean = ((番号 Mod 5) = 0)

        Dim s As String = 番号.ToString()
        If (is3の倍数 AndAlso is5の倍数) Then
            Return "Fizz Buzz"
        End If
        If (is3の倍数) Then
```

```
            Return "Fizz"
        End If
        If (is5の倍数) Then
            Return "Buzz"
        End If
        Return s
    End Function
```

ElseIf の連続が無くなって、スッキリしました。
途中で Return してしまうということは、「とある条件の時は、これ以降のコードを読まなくてよい」 ということですから、解読の負荷が小さくなります。 途中で Return させるのを嫌う人もいますが、このような使い方ならば逆に理解しやすくなることもあるのです。

それでは、忘れずに NUnit からグリーンを貰ってください。

§ 一時変数のインライン化 (Inline Temp)

おや…? ローカル変数 s には何の役割も無くなってしまいました。 この変数は消してしまいましょう。

```
    Private Shared Function Get発言(ByVal 番号 As Integer) As String
        Dim is3の倍数 As Boolean = ((番号 Mod 3) = 0)
        Dim is5の倍数 As Boolean = ((番号 Mod 5) = 0)

        'Dim s As String = 番号.ToString()
        If (is3の倍数 AndAlso is5の倍数) Then
            Return "Fizz Buzz"
        End If
        If (is3の倍数) Then
            Return "Fizz"
        End If
        If (is5の倍数) Then
            Return "Buzz"
```

```
        End If
        Return 番号.ToString()
    End Function
```

このリファクタリングでは、オマケとして、数字を返さないときのムダな ToString() メソッドの呼び出しが無くなりました。

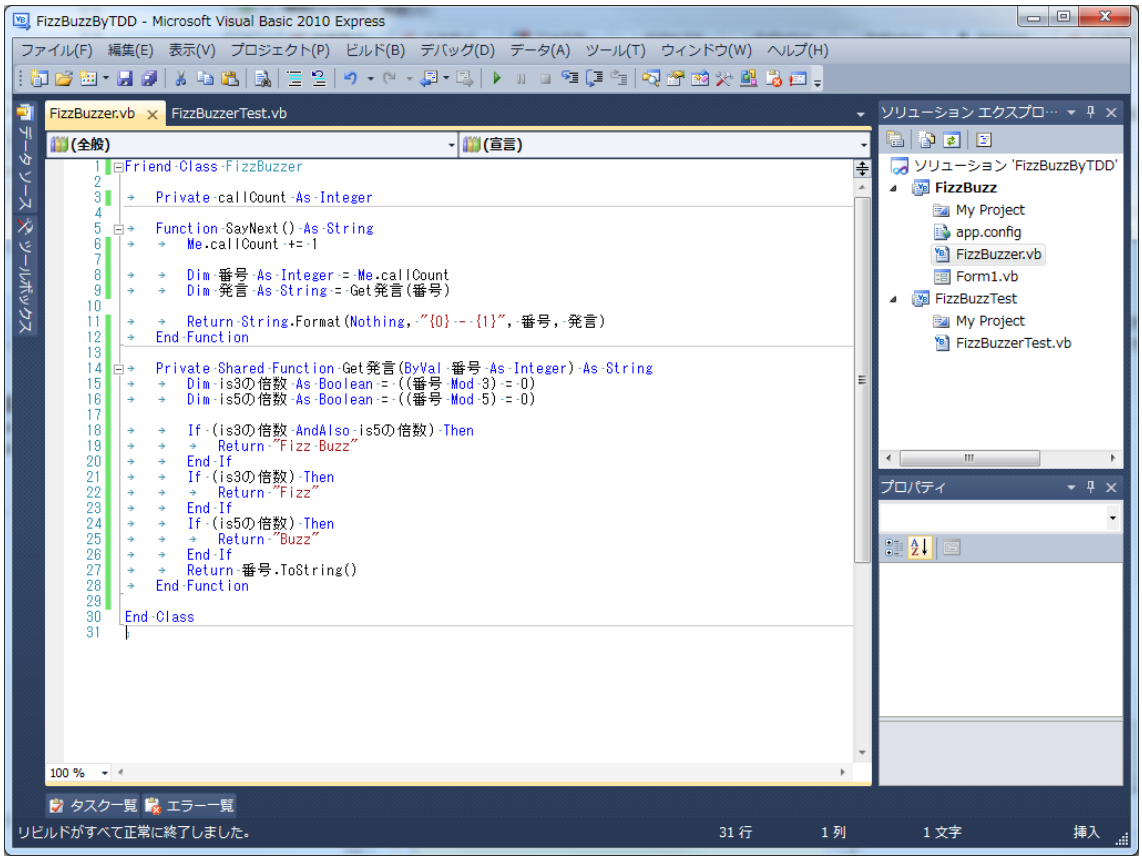
§ リファクタリング終了

これで、「4-3: リファクタリングする候補を挙げる」 に掲げたリファクタリング候補は、ほぼすべて実施しました。

製品コードは、このようになりました。

```
Friend Class FizzBuzzer

    Private callCount As Integer

    Function SayNext() As String
        Me.callCount += 1

        Dim 番号 As Integer = Me.callCount
        Dim 発言 As String = Get発言(番号)

        Return String.Format(Nothing, "{0} - {1}", 番号, 発言)
    End Function

    Private Shared Function Get発言(ByVal 番号 As Integer) As String
        Dim is3の倍数 As Boolean = ((番号 Mod 3) = 0)
        Dim is5の倍数 As Boolean = ((番号 Mod 5) = 0)

        If (is3の倍数 AndAlso is5の倍数) Then
            Return "Fizz Buzz"
        End If
```

```
        If (is3の倍数) Then
            Return "Fizz"
        End If
        If (is5の倍数) Then
            Return "Buzz"
        End If
        Return 番号.ToString()
    End Function
End Class
```



最後に、ビルドして、NUnit で全部のテストがグリーンになることを確認したら、リファクタリング終了です。

§ 付録: VB の Static キーワード

VB には、その長い歴史に由来する不思議なものがいろいろあります。 そのひとつが、ローカル変数に付けることのできる Static キーワードです。 これは、そのメソッドをスコープとするメンバー変数のようなものです。 Static キーワードを利用すると、SayNext() メソッドでしか使っていない callCount メンバー変数を、SayNext() メソッド内に移動できます。

よほど VB に詳しい人でないと知らない機能なので、実際の利用は避けた方がよいと思いますが、紹介しておきます。

```
Friend Class FizzBuzzer

    'Private callCount As Integer

    Function SayNext() As String
        Static callCount As Integer = 0
        callCount += 1

        Dim 番号 As Integer = callCount
        Dim 発言 As String = Get発言(番号)

        Return String.Format(Nothing, "{0} - {1}", 番号, 発言)
    End Function
```

信じられないかもしれませんが、ビルドして、NUnit でテストしてみると、ちゃんとグリーンになります。

☆ この時点でのソースコード一式 ⇒ 「FizzBuzzByTDD04.zip」 (26,504 バイト) をダウンロード http://www.tdd-net.jp/files/FizzBuzzByTDD04.zip (VB2010EE + NUnit2.5.9)

■ 5. システム テスト

◆ 5-1: GUI と結合して、プログラムの完成

それでは Windows フォームのコードに戻りましょう。 こうなっていました。

```
Public Class Form1

    Private Sub ButtonNext_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ButtonNext.Click
        'TODO: 後でロジックの呼び出しに置き換える
        Me.Label1.Text = "TEST"
    End Sub

End Class
```

FizzBuzzer はインスタンスを作らねばなりませんが、どこに書きましょう? このフォームがインスタンス化される時に、一緒に作ってしまってよさそうですね。 フォームのメンバー変数に宣言するとともに、New してしまいましょう。

そして、"TEST" のところをインスタンスの SayNext() メソッドに置き換えれば、ボタンをクリックするたびに順番に Fizz Buzz を表示してくれるでしょう。

```
Public Class Form1

    Private fizzBuzzer As FizzBuzzer = New FizzBuzzer()

    Private Sub ButtonNext_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ButtonNext.Click
        Me.Label1.Text = Me.fizzBuzzer.SayNext()
    End Sub
```

```
End Class
```

さぁ、F5 を押して実行してみましょう!

どうでしょうか、上手く動きましたか?

おっと残念、フォームが表示された時、デザイン画面のままです。 「1-1: 要件」 のところを見ると、最初は "1 - 1" と表示されねばなりません。

それには、Load イベント ハンドラーを追加します。 これは、フォームが最初に表示されたときに呼び出されるハンドラーです。 デザイナー画面を開き、フォームのタイトル部分をダブルクリックすると、ハンドラー メソッドが自動生成されます。

Load イベント ハンドラーにもボタンクリックのハンドラーと同じコードを書けば OK です。 ただ、同じコードが 2 箇所に出てくるのは面白くないので、メソッドへ切り出すリファクタリングをやっておくと、次のようになります。

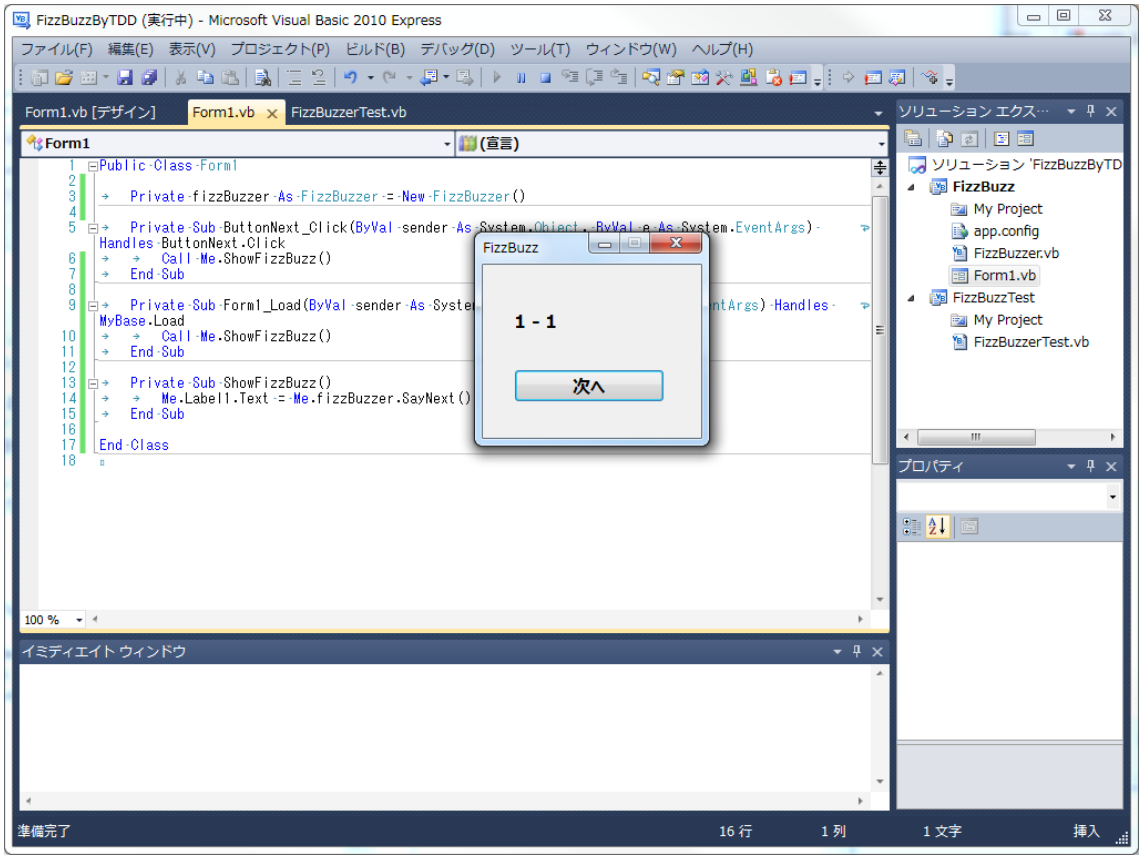
```
Public Class Form1

    Private fizzBuzzer As FizzBuzzer = New FizzBuzzer()

    Private Sub ButtonNext_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ButtonNext.Click
        Call Me.ShowFizzBuzz()
    End Sub

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Call Me.ShowFizzBuzz()
    End Sub

    Private Sub ShowFizzBuzz()
        Me.Label1.Text = Me.fizzBuzzer.SayNext()
    End Sub
```

```
End Class
```

VB の伝統のひとつに、サブルーチン (返値の無いメソッド) の呼び出しには Call キーワードを使う、というものがあります。 今では Call は省略可能ですが、付けてみました。

さぁ、こんどこそ!

実行して、要求通りに出来ていることを確かめてください。



☆ この時点でのソースコード一式 ⇒ 「FizzBuzzByTDD05.zip」 (26,025 バイト) をダウンロード http://www.tdd-net.jp/files/FizzBuzzByTDD05.zip (VB2010EE + NUnit2.5.9)

◆ 5-2: システムテスト

実装が完了しました。

これで製品として出荷して良いでしょうか? 答えはノーです。 従来通りにシステムテスト (結合テスト) に進みます。 システムテスト以降のことはここでは書きませんが、あくまでも TDD というのは詳細設計を含む実装作業を置き換えるだけのものだということを、覚えておいていただきたいと思います。

従来のテストが必要な理由を、考えておきましょう。

テストファーストによって、ロジック部分は全てのテストケースに合格しています。TDD 三原則を守っていたなら、テストのカバレッジは C0 100% になっているはずです。品質にはまったく問題が無いように思えます。

そこで見落としているのは、つぎのようなことです。

- **テストファーストで作成したテストケースが、プログラムの仕様と一致しているかどうか?**
- **最初に決めたプログラムの仕様そのものが (顧客にとって) 正しいかどうか?**

どちらも、メソッドレベルに注目している TDD では、答えることは不可能ですね。 やはり、システムテストは必要なのです。

ただし、システムテストで見つかる、つまらないバグは激減しているはずです。 TDD で実装に掛かる時間が数割長くなりますが、システムテストでのバグ潰しの時間が格段に減るので、トータルでは得になるでしょう。

※ ひとつめに挙げた理由をカバーするために、TDD を改良して BDD (振る舞い駆動開発) や ATDD (受け入れテスト駆動開発) といった技法が研究されています。 プログラム全体の仕様もテストコードとして記述してしまえというわけですが、GUI の見栄えや挙動をテストするところは難問です。